

Grant Agreement number: 101092696

Topic: HORIZON-CL4-2022-DATA-02



CODECO

Cognitive Decentralised
Edge Cloud Orchestration

D14: CODECO Federated Operation and Toolkit v2.0

Work package	WP4 – CODECO Federated Operation and Open Toolkit
Internal Number	D4.2
Task	Task 4.1, 4.2, 4.3, 4.4
Due date	28/02/2026
Submission date	30.03.2026
Dissemination Type	Public
Deliverable lead and editor	ICOM (G. Papathanail, Marinela Mertiri, Vasileios Theodorou)
Contributing Partners	FOR (Rute C. Sofia, Kaikang Huang); ICOM (George Papathanail, Marios Charalabides) ATH (George Koukis, Vassilis Tsaousidis); SIE (Jürgen Gesswein); I2CAT (Alejandro Espinosa, Rizkallah Tuma); UPRC (Efterpi Paraskevoulakou, Panagiotis Karamolegkos); TID (Alejandro Muniz, Luis M. Contreras Murillo); UC3M (Borja Nogales), UPM (Javier Serrano, Alberto del Rio, David Jimenez); RHT (Ray Carrol, Josh Salomon, Dean Kelly); IBM (Luis Garcés-Erice, Peter Urbanetz)
Revision version	0.9
Reviewer 1	R. C. Sofia, FOR
Reviewer 2	G. Koukis, ATH



**Funded by
the European Union**

Project funded by



Schweizerische Eidgenossenschaft
Confédération suisse
Confederazione Svizzera
Confederaziun svizra

Swiss Confederation

Federal Department of Economic Affairs,
Education and Research EAER
State Secretariat for Education,
Research and Innovation SERI

Project Partners

The fortiss logo consists of the word "fortiss" in a blue, lowercase, sans-serif font.The INOVA+ logo features the word "INOVA" in a bold, black, sans-serif font, followed by a plus sign.The Atos logo displays the word "Atos" in a blue, sans-serif font, with the 'A' and 'o' being larger and more prominent.The INTRACOM TELECOM logo features a red diamond icon with a white 'i' inside, followed by the words "INTRACOM" and "TELECOM" in a bold, black, sans-serif font.The ATHENA logo includes the word "ATHENA" in a bold, blue, sans-serif font, with "Research & Innovation" and "Information Technologies" in smaller text below it.The logo for GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN features a stylized 'G' icon followed by the university's name in a black, sans-serif font.The SIEMENS logo displays the word "SIEMENS" in a bold, green, sans-serif font.The netcompany intrasoft logo features the word "netcompany" in a bold, black, sans-serif font, with "intrasoft" in a smaller font below it.The ECLIPSE FOUNDATION logo includes the word "ECLIPSE" in a bold, black, sans-serif font, with "FOUNDATION" in a smaller font below it.The IBM logo features the word "IBM" in a bold, blue, sans-serif font, with the letters 'I' and 'M' being larger and more prominent.The i2cat logo features a stylized orange circular icon with dots, followed by the text "i2cat" in a bold, black, sans-serif font.The logo for the UNIVERSITY OF PIRAEUS RESEARCH CENTER features a stylized 'X' icon followed by the university's name in a black, sans-serif font.The Telefónica logo features a blue circular icon with dots, followed by the word "Telefónica" in a bold, black, sans-serif font.The logo for the UNIVERSIDAD POLITÉCNICA DE MADRID features a circular seal icon followed by the university's name in a black, sans-serif font.The Red Hat logo features a red hat icon followed by the words "Red Hat" in a bold, black, sans-serif font.The almende logo features a stylized network icon with dots and lines, followed by the word "almende" in a bold, black, sans-serif font, and "ORGANIZING NETWORKS" in smaller text below it.

Affiliated Entities

The GÖTTINGEN logo features a green 'G' icon followed by the word "GÖTTINGEN" in a bold, black, sans-serif font, and "STADT, DIE WISSEN SCHAFFT" in smaller text below it.The uc3m logo features the text "uc3m" in a bold, black, sans-serif font, followed by "Universidad Carlos III de Madrid" in a smaller font.The Atos EVIDEN logo features the word "Atos" in a bold, blue, sans-serif font, followed by "EVIDEN" in a bold, black, sans-serif font, and "an atos business" in smaller text below it.

Funded by
the European Union

Executive Summary

This report is an integral part of the CODECO deliverable *D14 – CODECO Federated Operation and Toolkit v2.0*. D14 consists of this report and the final CODECO software release, which provides the complete and stable version of the CODECO open-source *Federated Operation Toolkit v2.0*, available in the [CODECO Eclipse GitLab repository](#). This release corresponds to the final version of the code, focusing on the CODECO federated operation (CODECO FC OSS Toolkit) and completes the earlier prototype delivered in D13, which provided an early release of the CODECO OSS FC Toolkit.

D14 and the software developed are the result of the work conducted in CODECO Work Package 4 – CODECO Federated Operation and Open Toolkit – and represent the final output of the work developed across all tasks. The OSS integration was coordinated by T4.4 - Open Federated Edge-Cloud Toolkit Development, the task focusing on the implementation, integration, and testing of the CODECO Toolkit.

Keywords: Edge-Cloud continuum; federated orchestration; K8s; AI/ML; network; data; compute; context-awareness.

Document Revision History:

Version	Date	Description of change	List of contributor(s)
v0.1	11.11.2025	Proposal for a global ToC	George Papathanail (ICOM)
v0.2	10.02.2026	Input by all partners, CODECO	All partners involved
v0.3	27.02.2026	Input by FOR, section 2, section 3 (sub-components PDLC-CA, netma-nsm-mon)	Rute Sofia (FOR),
v0.4	04.03.2026	Input by RedHat, section 3, input by UC3M section 3 (NetMa), input by UPRC	Borja Nogales (UC3M), Dean Kelly (Redhat), Efterpi Paraskevoulakou (UPRC)
v0.5	05.03.2026	Input by ICOM section 3	Marios Charalambides (ICOM)
v0.6	09.03.2026	Internal revision 1	George Koukis (ATH)
v0.7	09.03.2026	Internal revision 2	Rute Sofia (FOR)
v0.8	11.03.2026	Verification by ICOM and release to coordinator	George Papathanail (ICOM)
v0.9	29.03.2026	Final review and release to EC	Rute C. Sofia (FOR)

Disclaimer:

The information, documentation and figures available in this deliverable have been developed by the Horizon Europe CODECO project consortium, under the European Union grant Agreement number 101092696. The content does not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.

Copyright notice: © 2023 - 2026 CODECO Consortium



Table of Contents

1	Introduction	10
1.1	Document Scope	10
1.2	Dependencies.....	11
1.3	Document Structure.....	11
2	CODECO Federated Operation Overview.....	11
2.1	Architectural Principles Overview.....	11
2.2	Framework Components	13
2.3	Governance, Coordination and Execution.....	15
2.4	Federated Workflow at Runtime	15
2.5	Management Layers and Federation Timescales	17
2.6	Energy Awareness	17
2.7	CODECO and the CEI Continuum.....	18
2.7.1	Runtime Example: AI across the CEI Continuum.....	19
2.8	CODECO OSS FC Toolkit Workflow Examples	19
2.8.1	Installation and Component Deployment	19
2.8.2	Deploying an Application Across a Federated CEI Environment.....	21
2.8.3	Runtime Management and Adaptation.....	22
2.8.4	Federation Scope and Partitioning.....	22
3	CODECO Components: Code Design and Structure	22
3.1	ACM-FC: Automated Configuration Management.....	23
3.1.1	Description.....	23
3.1.2	Sub-components.....	23
3.1.3	Code Structure	27
3.2	PDLC-FC: Privacy-preserving Decentralised Learning and Context-Awareness..	29
3.2.1	Description.....	29
3.2.2	Sub-components.....	30
3.3	NetMA-FC: Network Management and Adaptation	51
3.3.1	Component Description	51
3.3.2	Sub-components.....	52
3.4	MDM-FC: Metadata Manager	80
3.4.1	Final Component Description.....	80
3.4.2	Sub-components.....	81
3.5	SWM-FC: Scheduling and Workload Migration	93
3.5.1	Component Description	94
3.5.2	Sub-components.....	105

4	Continuous Integration, Testing, Deployment Preparation and Releasing	106
4.1	Testing Methodology	107
4.2	Deployment and CI/CD Methodology.....	108
4.2.1	Deployment	108
4.2.2	CI/CD Methodology	109
4.2.3	Integration and Testing Facilities	109
5	Conclusions, Takeaways, Extensions and Sustainability.....	110
5.1	Key Takeaways	110
5.2	Sustainability	111
5.3	Open Issues	111
5.4	Future Improvements and Extensions	112
6	References	113
7	Annex I: CODECO Metrics.....	114
8	Annex II – Release Versioning	119

List of Figures

Figure 1: the CODECO data-compute-network composite infrastructure view.....	12
Figure 2: The CODECO Framework modular and Kubernetes interoperable components. .	13
Figure 3: CODECO operation across three clusters, an OCM Hub cluster, and 2 OCM Managed Clusters.....	16
Figure 4: Example for phase 1, CODECO Operator deployment.....	20
Figure 5: PDLC-DP functional scheme.....	31
Figure 6: PDLC-CA workflow, FC operation.	35
Figure 7: PDLC-CA-FC, example for cost scoring defined for node resilience.....	36
Figure 8: Pre-defined target profile greenness, with equation defined by the user, based on CO2 foot printing.....	36
Figure 9: PDLC-CA-FC example for defining a custom target performance profile.....	37
Figure 10: PDLC-DL-GNNs functional scheme.	39
Figure 11: PDLC-DL-GNNs sequence diagram.....	41
Figure 12: PDLC-MARL-FC components sequence diagram.	48
Figure 13: NetMA internal workflows overview.	52
Figure 14: NetMA-FC Secure Connectivity Operational Workflow.....	54
Figure 15: netma-nsm-mon sequence diagram.....	61
Figure 16: netma-nsm-mon pod placement across control and service planes.....	61
Figure 17: example of output for the probes generated by netma-nsm-mon.....	67
Figure 18: Nemesis-FC functional scheme.....	72
Figure 19: Operation sequence, Nemesis-FC within a MC.....	73



Figure 20: Translation between ALTO maps and CODECO assets, nodes, clusters, links..	73
Figure 21: Unified multi-metric graph integrating overlay and underlying network data.	77
Figure 22: ALTO core interaction with the Federation API, illustrating the creation and exchange of local and federated network maps.	78
Figure 23: Operation sequence, Nemesys-FC within a MC	78
Figure 24: Translation between ALTO maps and CODECO assets, nodes, clusters, links, etc.	79
Figure 25: MDM sub-components and APIs to other components.....	80
Figure 26: Interaction between MDM subcomponents.....	88
Figure 27: A neighborhood with three clusters connected by an L2SM network	94
Figure 28: An application with workloads in two clusters.	95
Figure 29: L1 problem and infrastructure of cluster C_1 yield the L2 problem to solve in C_1 ..	97
Figure 30: Event-sequence diagram for placement of an application.	98
Figure 31: A channel connecting workloads into two clusters has three parts as far as the latency is concerned.	99
Figure 32: Topology of the infrastructure for the example placement.	101

List of Tables

Table 1: Key features of the GUI sub-component.....	24
Table 2: Main specification fields of the CodecoApp CRD.....	28
Table 3: Workload (microservice) fields defined in the CodecoApp specification.....	28
Table 4: Channel configuration fields and QoS parameters for service communication.....	29
Table 5: PDLC-DP sub-component source code.....	33
Table 6: Summary of main PDLC-CA source code files.	38
Table 7: PDLC-DL-GNN sub-components source code	44
Table 8: PDLC-FC-MARL modes	47
Table 9: PDLC-FC-MARL code structure	48
Table 10: Secure-Connectivity FC code structure	55
Table 11: Key differences on the update from active to passive probing.	59
Table 12: Description of the main netma-nsm-mon source code files.....	69
Table 13: Description of the main netma-nsm-npp source code files.....	71
Table 14: Nemesys-FC main code files.....	74
Table 15: of the main MDM GraphDB source code files.....	81
Table 16: Description of the main MDM Controller source code files.....	83
Table 17: Description of the main MDM API source code files.	83
Table 18: Main MDM K8s connector source code files.....	90
Table 19: Description of the main MDM Compliance connector source code files.....	91



Table 20: Description of the main MDM K8s connector source code files. 92

Table 21: CODECO Application Model Attributes, spec, and status, minimum subset of parameters..... 114

Table 22: CODECO metrics being collected by MDM 116

Table 23: CODECO metrics being collected by NetMA. 117

Table 24: CODECO sub-components, URLs, and open-source releases..... 119

List of Acronyms and Definitions

Acronym	Meaning
ACM	Automated Configuration Manager
AD	Anomaly Detection
AI	Artificial Intelligence
ALTO	Application-Layer Traffic Optimization
API	Application Programming Interface
APOC	Awesome Procedures on Cypher
ARM	Advanced RISC Machines
CA	Context-Awareness
CAM	CODECO Application Model
CDN	Content Deliver Network
CEI	Cloud-Edge-IoT
CP	Change Point
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command Line Interface
CNI	Container Network Interface
CODECO	Cognitive, Decentralised Edge-Cloud Orchestration
CODEF	CODECO Experimentation Framework
CPU	Central Processing Unit
CR	Custom Resource
CRD	Custom Resource Definition
DBMS	Database Management System
DEV	Developer
DL	Decentralised Learning
DL	Deep Learning
eBPF	Extended Berkeley Packet Filter
EC	European Commission
FC	Federated Cluster
FedGNNs	Federated GNNs
FIDO	Fast Identity Online
GNNs	Graph Neural Networks
GUI	Graphical User Interface
HE	Horizon Europe
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPAM	IP Address Management
JSON	Javascript Object Notation
K8s	K8s
KCP	K8s like Control Plane
KFP	Kubeflow Pipelines
KinD	K8s in Docker
L2	Layer 2
L2S-M	Link-Layer Secure connectivity for Microservice platforms
LPM	L2S-M Performance Measurements
MAPE	Mean Absolute Percentage Error
MARL	Multi Agent Reinforcement Learning
MDM	Metadata Manager

MEC	Multi-Access Edge Computing
MGT	Infrastructure Manager
ML	Machine Learning
MLOps	Machine Learning Operations
MSE	Mean Squared Error
NetMA	Network Management and Adaptation
NSM	Network State Monitoring
OAS	Open API Specification
OCM	Open Cluster Management
ONOS	Open Network Operating System
OS	Operating System
OSS	Open-Source Software
PDLC	Privacy-preserving Decentralised Learning and Context-awareness
POC, PoC	Proof of Concept
PP	Preprocessing
PVC	Persistent Volume Claim
PPO	Proximal Policy Optimization
QoE	Quality on Experience
QoS	Quality of Service
REST	Representational State Transfer
RL	Reinforcement Learning
RPC	Remote Procedure Call
SC	Single Cluster
SDG	Synthetic Data Generator
SDK	Software Development Kit
SDN	Software Defined Network
STGNN	Spatio-Temporal Graph Neural Network
SWM	Scheduling and Workload Migration
TEE	Trusted Execution Environment
UI	User Interface
UML	Unified Modelling Language
VLAN	Virtual Local Area Network
VM	Virtual Machine
YAML	Yet Another Markup Language

Acknowledgements

We thank all CODECO Partners involved in Work Package 4 for the commitment and involvement in the development of Deliverable D14.



1 Introduction

CODECO Work Package 4 (WP4) builds on the foundations established in WP3 to extend the CODECO framework beyond single-cluster orchestration into a federated, multi-cluster environment. From M19 through M39 (March 2026), WP4 has focused on the design, development, and validation of the CODECO federated architecture, enabling decentralised coordination across geographically and administratively distinct clusters. This work adheres to the architectural principles and component interfaces defined in earlier deliverables — specifically D10 (M18), D12 (M18), and D31 (M39).

Deliverable D14 – CODECO Federated Operation and Toolkit v2.0 presents the final CODECO Federated Cluster (FC) Operation Toolkit, designed to support application deployment and runtime management across multi-cluster, federated IoT-Edge-Cloud environments. The deliverable comprises this technical report alongside a Technology Readiness Level (TRL) 4–5 software release published under the CODECO Eclipse GitLab Research project¹. It concludes the implementation of CODECO's federated capabilities, encompassing inter-cluster communication, distributed lifecycle management, and cooperative resource scheduling.

Tasks 4.1 through 4.3 cover the specification, implementation, and integration of CODECO federated components with the broader architecture. Task 4.4 addresses end-to-end system integration and validation of the federated toolkit, spanning all WP4 technical tasks. As in prior iterations of the CODECO framework, WP4 follows an agile development methodology, enabling continuous refinement of components and their coordination as the federated architecture matured.

The objective of D14 is to deliver the final version of the CODECO OSS Federated Toolkit, completing the work initiated in D13 and consolidating a validated set of federated components in line with the project's open-source development strategy. D14 therefore consists of this report and the associated codebase hosted on the CODECO Eclipse GitLab. Development efforts are currently transitioning to the operational Eclipse technology KuDECO project².

1.1 Document Scope

D14 refers to the final release of the open-source **CODECO Federated Clusters Toolkit v2.0**, along with an explanation of its software architecture and implementation design. This deliverable builds on the previous versions of the CODECO Basic Operation Toolkit (D12), completing the framework to support federated clusters operation. This deliverable provides:

- An explanation of the federated CODECO framework and final implementation procedures for the Federated Clusters Toolkit.
- A description of the tools and methods adopted to support the implementation workflow, including coding guidelines, installation instructions, and the core tools and programming languages used in the development of the federated CODECO components.

D14 is composed of the following parts:

- This report (software companion report).
- The final release of the CODECO Federated Cluster Operation Toolkit v2.0, available via the [CODECO Eclipse GitLab](https://projects.eclipse.org/projects/technology.kudeco), and currently under transfer to the Eclipse technology project KuDECO.

¹ <https://projects.eclipse.org/projects/technology.kudeco>

² <https://projects.eclipse.org/projects/technology.kudeco>

1.2 Dependencies

D14 is a companion report to the CODECO Federated Operation OSS Toolkit v2.0 provided in the Eclipse Gitlab Research Repository of the project, and being transferred to the operational Eclipse KuDECO project.

D14 is a self-contained document, intended to be used by users that want to experiment or use CODECO as developers. For users that want to understand the overall architectural principles and design of CODECO, D31 [2] should be considered.

1.3 Document Structure

- **Section 1** introduces the overall deliverable scope, structure, and components.
- **Section 2** provides a summary of the CODECO architectural design and highlights the final set of sub-components included in the final release.
- **Section 3** provides the companion reporting, the use of the final codebase, detailed components and sub-components.
- **Section 4** describes the continuous system testing and deployment framework established in T4.4 support the final validation of the project, as well as the continuous integration and released version of each sub-component/feature delivered in this final implementation cycle of CODECO.
- **Section 5** summarizes the report, providing key takeaways.

2 CODECO Federated Operation Overview

This section introduces CODECO from the perspective of a Kubernetes developer encountering the framework for the first time. You are assumed to be familiar with Kubernetes concepts such as CRDs, controllers, reconciliation loops, multi-cluster management, and observability stacks. What CODECO adds is a structured way to perform federated, context-aware, and application-centric orchestration across heterogeneous Edge–Cloud environments.

The detailed architectural rationale and blueprint are described in Deliverable D31. The purpose of this section is to summarize the implemented architecture and explain how its components interact at runtime.

CODECO extends Kubernetes into a federated orchestration framework for the *Compute–Edge–Intelligence (CEI)* continuum. Instead of treating clusters as independent scheduling islands, CODECO treats applications as distributed entities that may span clusters, evolve over time, and adapt dynamically to changes in infrastructure, network conditions, data locality, user behaviour, and energy constraints.

2.1 Architectural Principles Overview

At its foundation, CODECO remains Kubernetes-native. All enforcement ultimately resolves into Kubernetes manifests applied to *Managed Clusters*. Federation and context-awareness are layered on top of existing Kubernetes control patterns rather than replacing them.

Federation is implemented using *Open Cluster Management (OCM)*³, which provides the hub-and-spoke substrate for cluster registration, identity management, and policy propagation.

³ <https://open-cluster-management.io/>

CODECO builds additional logic on top of this substrate to support application-aware coordination, neighbourhood scoping, AI-assisted decision support, and cross-layer observability.

A key conceptual difference from standard Kubernetes scheduling is that CODECO does not consider compute resources in isolation. Instead, it adopts a **data-compute-network** infrastructure view, represented in Figure 1.

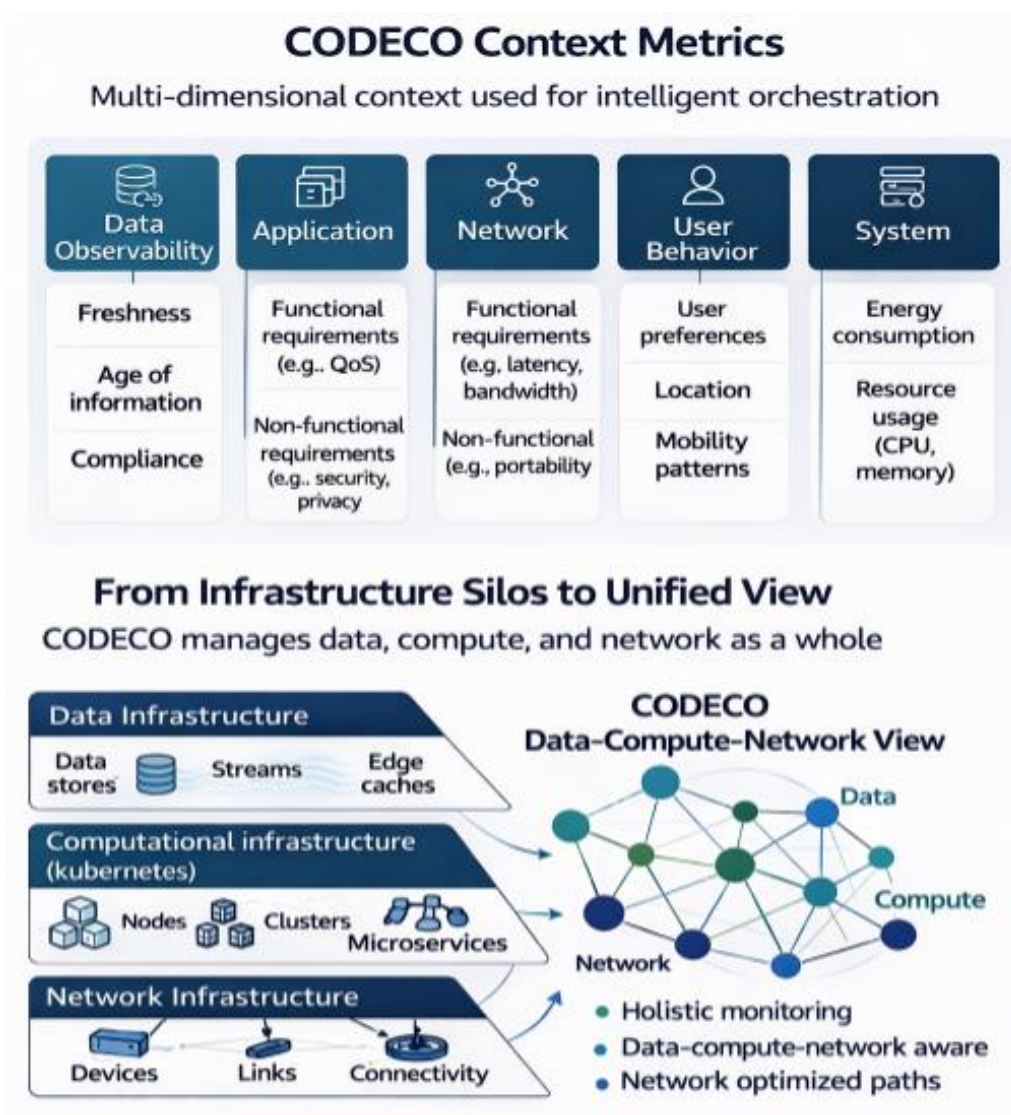


Figure 1: the CODECO data-compute-network composite infrastructure view.

The system models three infrastructure planes simultaneously:

- Data infrastructure (storage, processing, locality, compliance).
- Computational infrastructure (CPU, memory, energy).
- Network infrastructure (underlay and overlay perspective).

These control plane layers are not independent. Placement decisions must account for their interactions. For example, moving an application micro-service may improve compute utilization but violate data locality constraints or introduce unacceptable latency due to network conditions.

To support this integrated view, CODECO continuously collects structured observability signals across multiple dimensions. The metrics consumed by orchestration logic span five main categories, represented in Figure 1:

- Data observability metrics describe freshness, age of information, and compliance constraints. Application metrics capture functional (QoS) and non-functional (security, privacy) requirements.
- Network metrics describe QoS properties and portability constraints. User behaviour metrics capture user defined input and preferences. System metrics include CPU utilization and energy consumption. These dimensions are normalized and injected into the orchestration pipeline. Unlike a standard Horizontal Pod Autoscaler reacting to CPU thresholds, CODECO evaluates composite cost and stability indicators derived from cross-layer context.

2.2 Framework Components

At a high level, CODECO decomposes orchestration into distinct management functions implemented as Kubernetes-native services (controllers, operators, agents, and CRDs). Each component has a clearly bounded responsibility and interacts with others through declarative interfaces and reconciliation loops. The functional components are represented in Figure 2.

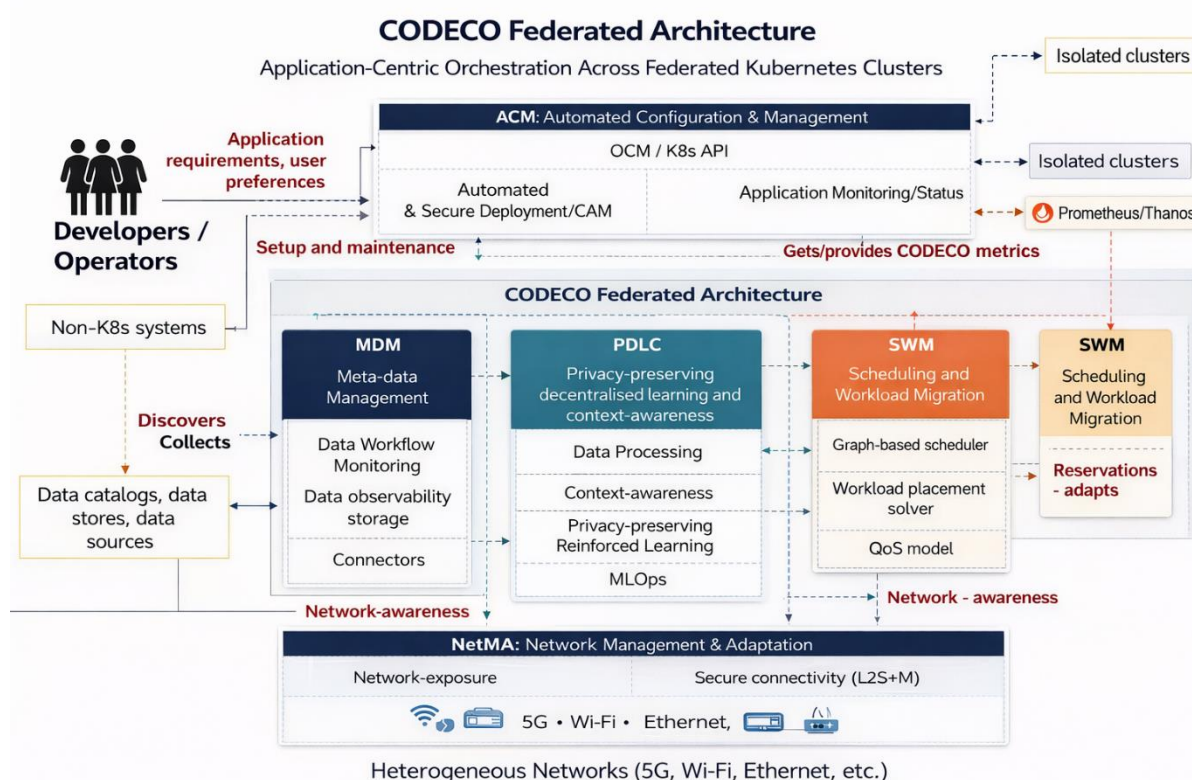


Figure 2: The CODECO Framework modular and Kubernetes interoperable components.

ACM-FC is the primary entry point for developers and operators. It consumes the CODECO Application Model (CAM), interprets application intent, and translates it into actionable federation logic. From an implementation perspective, ACM-FC:

- Parses the CAM specification.
- Validates intent and policy constraints.

- Triggers neighbourhood computation.
- Coordinates application lifecycle events.
- Maintains the application status view across clusters.

ACM-FC runs centrally in the Hub cluster and deploys lightweight agents into Managed Clusters. The Hub instance operates at the governance layer, while agents participate in execution-layer reconciliation. ACM-FC is responsible for defining *what* should happen, not *where* it should ultimately run.

PDLC-FC provides AI-assisted recommendation and context aggregation. It analyzes cross-layer metrics collected from compute, network, and data infrastructure and produces cost and stability indicators that inform scheduling decisions. PDLC-FC operates primarily within the coordination layer and is instantiated in Managed Clusters that belong to a given neighbourhood. This ensures that learning and optimization remain bounded and scalable.

PDLC-FC does not execute placement actions. Its outputs are advisory signals consumed by SWM-FC. This separation guarantees that policy enforcement remains deterministic and auditable.

NetMA-FC supplies the network perspective required for federated orchestration. It exposes observability metrics and configures secure inter-cluster connectivity.

Its responsibilities include:

- Collecting QoS metrics such as latency and bandwidth.
- Monitoring topology changes and mobility effects.
- Establishing secure overlay connectivity across clusters.
- Exposing normalized network capabilities to scheduling components.

NetMA-FC spans the federation, with monitoring agents close to workloads and coordination logic at the Hub level. It ensures that placement decisions account for network conditions rather than assuming static connectivity.

MDM-FC provides the data-awareness layer of CODECO. It maintains a metadata graph that captures:

- Data locality and ownership.
- Freshness and age-of-information.
- Compliance and portability constraints.
- Dataflow dependencies among microservices.

This metadata graph is exposed through APIs consumed by orchestration components. Placement and migration decisions therefore incorporate data governance and locality constraints alongside compute and network metrics. MDM-FC instances are typically scoped per neighbourhood, limiting metadata exchange to application-relevant clusters and avoiding global state dissemination.

SWM-FC is responsible for enforcing placement and migration decisions across the federation. It bridges advisory decision support and Kubernetes-native scheduling enforcement.

The implementation is split into two logical layers:

- SWM-L1, hosted in the Hub cluster, coordinates global placement decisions within the application neighbourhood.

- SWM-L2, deployed in each Managed Cluster, enforces local scheduling, migration, and redeployment actions.

SWM-FC consumes application constraints from ACM-FC and contextual indicators from learning components. It evaluates candidate clusters within the bounded neighbourhood scope and produces placement directives that are translated into Kubernetes manifests. All enforcement remains reconciliation-driven and policy-filtered.

2.3 Governance, Coordination and Execution

The architecture is intentionally layered to balance central authority with edge autonomy. As presented in Figure 3, the governance layer runs in the Hub cluster and defines desired state, application intent, and global policy constraints. The Hub is authoritative but not continuously controlling. Policies and intent are propagated through OCM, ensuring consistent compliance across clusters.

The coordination layer scopes optimization through application **neighbourhoods**. Instead of reasoning over all clusters in the federation, CODECO computes **a bounded subset of clusters relevant to a specific application**. Monitoring exchange, learning collaboration, and scheduling optimization are restricted to this neighbourhood. This design avoids global state explosion and keeps multi-cluster optimization tractable.

The execution layer resides in Managed Clusters. Local agents enforce placement and migration decisions using Kubernetes-native mechanisms. Importantly, execution autonomy is preserved during intermittent Hub connectivity. When the Hub becomes reachable again, reconciliation ensures convergence toward the globally defined desired state.

2.4 Federated Workflow at Runtime

The federated workflow begins when a developer submits an application described using the CODECO Application Model (CAM) abstraction (rf. To section 3.1). CAM extends the declarative Kubernetes style with semantic intent, including latency targets, energy preferences, resilience objectives, and compliance requirements.

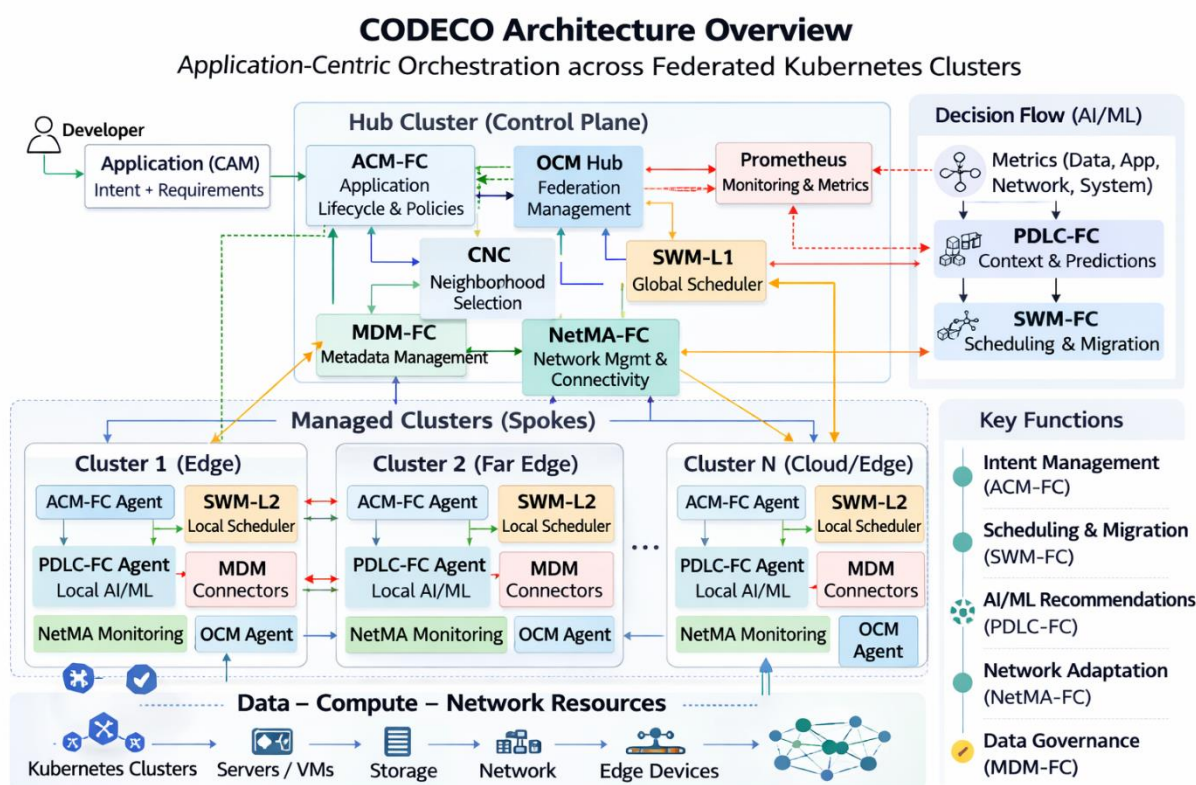


Figure 3: CODECO operation across three clusters, an OCM Hub cluster, and 2 OCM Managed Clusters.

As illustrated in Figure 3, ACM-FC interprets the CAM and triggers neighbourhood computation. The *Neighbourhood Composer* (CNC) selects a bounded set of candidate clusters based on infrastructure characteristics and application requirements. Within the federated example provided in Figure 3:

- The Hub hosts ACM-FC, SWM-L1, OCM Hub, and central observability components.
- Managed Clusters host ACM agents, SWM-L2, PDLC agents, NetMA monitoring, and MDM connectors.
- Monitoring data flows upward; scheduling decisions flow downward.

Observability signals from networking (NetMA-FC), metadata (MDM-FC), and system metrics (Prometheus) are aggregated within the neighbourhood scope. PDLC-FC analyses these signals and produces advisory cost and stability indicators. These outputs do not directly trigger scheduling actions. Instead, SWM-FC integrates them with policy constraints and real-time cluster state to compute placement or migration decisions.

Enforcement occurs through Kubernetes-native primitives in each Managed Cluster. Network reconfiguration is delegated to NetMA-FC, and metadata updates propagate through MDM-FC. The system then returns to a steady reconciliation state until new context changes trigger re-evaluation.

Throughout this process, policy remains authoritative, and AI remains advisory. The system is explicitly designed to avoid uncontrolled automation that could violate compliance or governance constraints.

2.5 Management Layers and Federation Timescales

To maintain both global consistency and local responsiveness, CODECO structures decision-making across three temporal layers.

The **governance layer** operates at longer timescales and defines application intent, policy constraints, and federation membership. Components at this layer, such as ACM-FC in the Hub and the federation substrate, establish the desired state of the system and enforce organizational or regulatory rules.

The **coordination layer** operates at medium timescales. It scopes orchestration through application neighbourhoods, aggregates observability signals, and computes adaptive recommendations. PDL-FC and SWM-L1 primarily operate at this layer, translating context into bounded optimization decisions without violating governance constraints.

The **execution layer** operates at short timescales inside Managed Clusters. SWM-L2, NetMA monitoring agents, and ACM-FC agents enforce placement, migration, and network configuration using fresh local context. Execution remains autonomous under intermittent Hub connectivity. When connectivity is restored, reconciliation ensures convergence toward the desired global state.

By distributing components across these layers, CODECO avoids both rigid centralization and uncontrolled decentralization. Governance remains authoritative, coordination remains bounded and scalable, and execution remains resilient and responsive. For developers, this layered separation provides clear extension points and predictable control boundaries when integrating new capabilities or modifying orchestration logic.

2.6 Energy Awareness

Energy monitoring in CODECO is not limited to a single metric or layer. It is implemented as a multi-level model that captures energy impact across compute, network, and application flows. This comprehensive approach enables the orchestration layer to reason about sustainability and operational cost in a federated CEI environment without reducing energy awareness to a simplistic proxy.

At the infrastructure level, **node energy** reflects the energy consumption associated with compute activity on a given node (collected via ACM-FC). This includes CPU utilization, memory pressure, and potentially accelerator usage where available. Node energy provides a coarse-grained estimate of how “expensive” it is to execute workloads on a specific cluster or node. In energy-aware scheduling scenarios, this allows PDL-FC to differentiate between lightly loaded edge nodes, energy-constrained devices, and high-capacity cloud resources.

At the network level, **link energy** (collected by NetMA, netma-nsm-mon) represents the energy cost of traffic exchanged between two nodes. It aggregates the observed byte volume over a link and applies calibrated energy-per-byte models. Link energy provides visibility into the cost of moving data across the infrastructure and becomes particularly relevant in geographically distributed or bandwidth-constrained federations. It enables reasoning about the energy impact of cross-cluster communication rather than assuming network transport is neutral.

At a finer granularity, **flow energy** (collected by NetMA, netma-nsm-mon) captures the energy footprint of specific application data flows (IP flows). Unlike link energy, which aggregates all traffic between nodes, flow energy isolates the contribution of individual microservices or application groups. This makes it possible to evaluate the energy implications of architectural decisions at the application level. For example, splitting a pipeline across clusters may increase total flow energy due to repeated cross-cluster transfers, even if compute utilization appears balanced.

These energy dimensions support greenness-aware scheduling and cost modelling within PDLC. Placement decisions are no longer based solely on performance or resource utilization; they can incorporate energy cost as a first-class optimization parameter. This enables energy-aware placement strategies, such as preferring local inference at the edge to reduce backhaul traffic, or centralizing compute-intensive workloads when data movement is limited.

In federated CEI environments, application placement directly influences data movement volume, network traversal distance, and cross-cluster traffic intensity. Energy monitoring across node, link, and flow levels allows PDLC to reason about whether migrating a microservice increases total communication energy cost, whether distributing workloads reduces aggregate energy consumption, or whether centralized training is more energy-efficient than decentralized processing.

Because the current implementation for energy monitoring relies on passive eBPF-based monitoring, all energy metrics are derived from real observed traffic and runtime behaviour rather than synthetic probes. This ensures that energy-aware scheduling decisions are grounded in operational conditions, making sustainability objectives measurable, comparable, and enforceable across heterogeneous clusters.

2.7 CODECO and the CEI Continuum

CODECO operationalizes the CEI continuum by treating heterogeneous infrastructure domains as first-class, federated execution environments under a unified control model. Rather than encoding Cloud, Edge, or IoT (far Edge) as rigid tiers in the control plane, CODECO abstracts each environment as a Hub-Spoke described through capability metadata, policy constraints, and contextual signals.

From the orchestration perspective, the continuum is therefore not a hierarchy but a **capability space**. Clusters differ in characteristics, not in orchestration semantics.

Edge clusters typically advertise:

- Low-latency proximity to data sources and users
- Limited compute and storage capacity
- Higher volatility (mobility, intermittent connectivity)

Cloud clusters typically advertise:

- Large compute pools and accelerator availability (e.g., GPUs)
- Elastic scaling properties
- Higher network distance from data sources

Far Edge clusters emphasize:

- Data sovereignty and regulatory compliance
- Controlled connectivity boundaries
- Organizational governance constraints

These properties are encoded in cluster descriptors and continuously updated through observability pipelines (compute, network, data, energy). Scheduling logic evaluates these attributes dynamically. No placement rule is hard-coded to “cloud” or “edge”; instead, placement emerges from policy filtering and multi-dimensional scoring.

2.7.1 Runtime Example: AI across the CEI Continuum

Let us assume an AI application to be deployed across CEI, with the following aspects:

- **Edge: Data Collection & Preprocessing.** The application is initially deployed in an edge cluster close to sensors or users. Low-latency access and data locality dominate placement decisions.
- **Cloud: Model Training.** When training requires computational resources unavailable at the Edge, CODECO would check suitable cluster candidates, and a Cloud cluster with sufficient processing capability would be selected. The training workload is federated without altering the CAM specification.
- **Edge: Inference Deployment.** The trained model artifact is redistributed to Edge clusters for inference, restoring proximity to users and minimizing latency.

Throughout this lifecycle, the declarative CAM remains unchanged. Federation decisions are policy-compliant, and placement (SWM-FC) is driven by context-aware scoring and AI recommendations. Enforcement remains Kubernetes-native and reconciliation-driven.

For developers, the key architectural point is that CODECO extends Kubernetes without breaking its mental model. The same declarative specification can span multiple infrastructure domains. Federation, neighbourhood scoping, and context integration operate beneath the application abstraction. The continuum becomes an optimization surface rather than an operational boundary.

2.8 CODECO OSS FC Toolkit Workflow Examples

This section provides the reader (developer) with a view on the practical lifecycle of the CODECO FC Toolkit, from installation to runtime orchestration. The workflows described here correspond to the communication sequences illustrated in Figure 5 and Figure 6.

From a developer perspective, the important shift from the Basic Toolkit is not conceptual, it is **scoping and federation-aware execution**. The application model (CAM) remains declarative and unchanged. What changes is how placement, optimization, and coordination operate across clusters instead of within a single cluster.

2.8.1 Installation and Component Deployment

The Federated Toolkit requires:

- One Hub cluster
- One or more Managed Clusters (MCs)
- Federation substrate provided by OCM

All CODECO components are deployed using standard Kubernetes manifests and Helm charts. Inter-component communication is implemented using CRDs. There are no side channels: coordination is CR-driven and reconciliation-based.

The developer (DEV) first prepares the federated Kubernetes infrastructure: one Hub and multiple Managed Clusters (e.g., MC1, MC2). The deployment sequence represented in Figure 4 is initiated via the ACM-FC interface (GUI or CLI). When the CODECO Operator is deployed on the Hub:

- The **CAM Engine** is initialized for application model parsing.
- The CNC (Neighbourhood Composer) plugin is registered.

- Federation interfaces to OCM are activated.

At this stage, the Hub becomes the governance authority for CODECO components. Once the operator is running, ACM-FC deploys CODECO components across the federation.

The Hub cluster hosts:

- **SWM-L1** — global scheduler and placement coordinator
- **NetMA (Hub controller)** — overlay coordination
- **MDM-FC Core** — metadata graph and data-awareness service

These components operate primarily in the governance and coordination layers.

Each MC shall have:

- **SWM-L2** — local scheduler enforcement agent
- **NetMA-FC agents** — network probing and secure overlay logic
- **PDLC-FC agent** — decentralized learning and context aggregation
- **MDM Connectors** — bridge between cluster-local data sources and the MDM Hub

All components interact through CRDs propagated via OCM. State propagation occurs through declarative updates. After deployment, the monitoring pipeline is immediately active.

- **NetMA-FC** continuously and passively probes network paths (latency, jitter, energy). Results are published as CRs and exposed via Prometheus.
- **MDM Connectors** discover and synchronize metadata (data locality, compliance, freshness) and update the metadata graph.
- **ACM-FC** monitors application-level metrics and user-defined preferences.

This monitoring loop feeds PDLC and SWM continuously and drives runtime adaptation.

CODECO Multi-Cluster Deployment Sequence

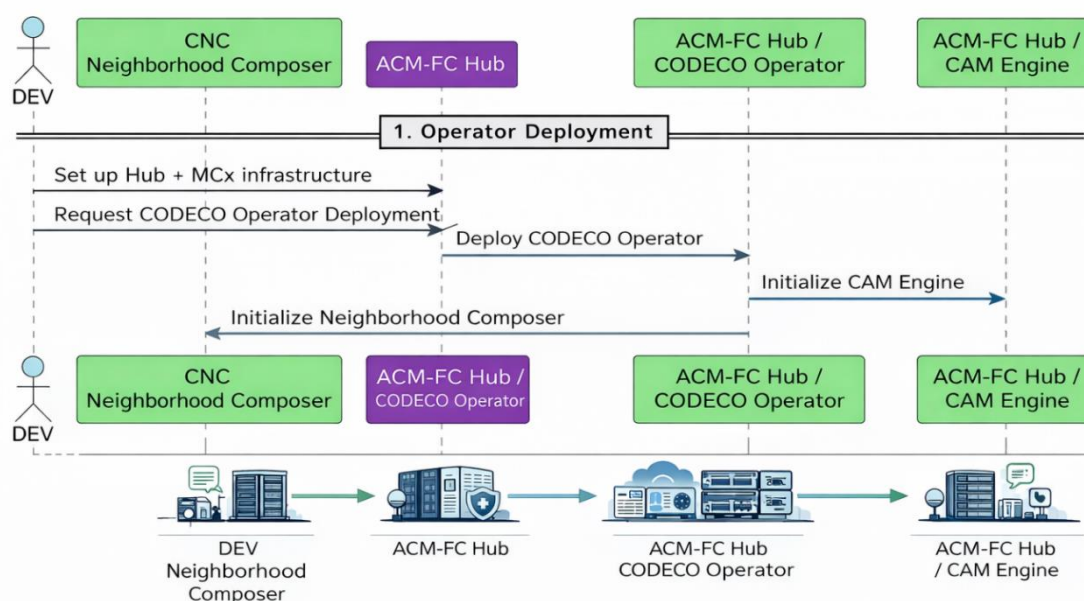


Figure 4: Example for phase 1, CODECO Operator deployment.

2.8.2 Deploying an Application Across a Federated CEI Environment

Application deployment in the federated toolkit follows the same declarative pattern as Kubernetes. The developer DEV prepares a CAM in YAML (or via the ACM GUI). CAM defines:

- Microservices
- Runtime requirements (QoS, QoE)
- Non-functional constraints (energy, compliance)
- Performance objectives

The developer submits the CAM to ACM-FC. From this point onward, orchestration is automatic.

Neighbourhood selection

Upon receiving the CAM:

1. ACM-FC queries the **CNC plugin**.
2. CNC computes a bounded set of Managed Clusters suitable for the application.
3. The resulting neighbourhood is fixed for deployment and periodically re-evaluated.

This bounded search space is critical. Without it, global optimization across all clusters would be computationally expensive and operationally unstable.

Placement Estimation

ACM translates the CAM into the internal SWM-L1 Application Model. In parallel:

- PDLC-FC agents are instantiated within the neighbourhood.
- PDLC (PDLC-DP) reads:
 - CAM constraints
 - Network metrics from NetMA (via Prometheus)
 - Compute metrics via Prometheus
 - Data metadata from MDM

PDLC (**PDLC-CA**) computes context-aware cost estimates for each candidate cluster and node. These estimates are written as annotated node costs into the SWM CRD *AssignmentPlan*. It is important to note that PDLC **does not execute placement**. It annotates and recommends.

Scheduling and enforcement

SWM-L1 evaluates the annotated costs and computes the final placement decision.

The placement decision is written into:

- AssignmentPlan
- ApplicationGroup CRs

SWM-L2 agents in each MC reconcile these CRs and enforce deployment using native Kubernetes scheduling mechanisms. NetMA configures secure overlays if cross-cluster communication is required. At this point, the application is running across the federated neighbourhood.

2.8.3 Runtime Management and Adaptation

Once deployed, CODECO enters steady-state runtime support. Assume:

- Microservice **a1** runs on MC1
- Microservice **a2** runs on MC2
- Both clusters are registered to the same Hub

The application is now fully federated but scoped within one neighbourhood under one Hub.

If runtime conditions change, e.g., resource exhaustion, network degradation, or there are energy constraints (triggers), then ACM-FC, following regular Kubernetes mechanisms, triggers re-evaluation. The sequence is:

1. PDLC recalculates context-aware behavior estimates
2. Updated costs are written to SWM via CRs
3. SWM recomputes placement
4. Updated AssignmentPlan CRs are written
5. SWM-L2 enforces the new placement

There is no direct feedback channel between PDLC and SWM. CRDs themselves serve as the coordination mechanism. This design ensures deterministic reconciliation and auditability, while keeping loose coupling.

2.8.4 Federation Scope and Partitioning

By default, CODECO evaluates resources within all MCs under a Hub. However, evaluating every cluster for every application is expensive in large-scale environments. The **Neighbourhood Composer (CNC)** reduces this complexity by:

- Pre-partitioning clusters using similarity metrics
- Restricting optimization to relevant subsets
- Limiting monitoring exchange to application-scoped clusters

This partitioning approach is expected to assist in reducing scheduling latency and the overall optimization complexity without sacrificing adaptability.

3 CODECO Components: Code Design and Structure

The present section provides a comprehensive description of the FC versions of the CODECO components, in particular, ACM, PDLC, NetMA, MDM and SWM. The description of the components is provided as companion to the code, and aims at a developer audience. Therefore, for each component the sub-components, their communication sequence, and code structure is described.

3.1 ACM-FC: Automated Configuration Management

3.1.1 Description

[ACM-FC](#) is the main entry point and orchestrator for the CODECO (Cognitive Decentralized Edge Cloud Orchestration) platform. Developed by Red Hat, ACM-FC serves as a Kubernetes operator that provides:

- **Application Lifecycle Management:** ACM receives application definitions via the CodecoApp Custom Resource Definition (CRD) and transforms them into deployable SWM (Scheduling and Workload Migration) applications.
- **Cross-Component Orchestration:** ACM orchestrates the deployment and configuration of all other CODECO components including SWM, MDM (Metadata Management), PDLC (Privacy-preserving Decentralized Learning and Context-awareness), and NetMA (Network Management and Adaptation).
- **Metrics Collection and Status Management:** ACM queries Prometheus for application, service, and node-level metrics and populates the CodecoApp status with near real-time performance data.

Two Deployment Paradigms:

- **Kubernetes (K8s) Deployment:** Traditional containerized workloads on Kubernetes clusters
- **Non-Kubernetes (Non-K8s) Deployment:** Edge device deployments via *Flightctl* using Virtual *Kubelet*.

User Interfaces:

- **API/CRD Interface:** For programmatic deployments.
- **Web GUI:** React/PatternFly-based interface for YAML generation, resource management, and monitoring.

3.1.2 Sub-components

3.1.2.1 ACM Operator (Core Controller)

Location: [controllers/codecoapp_controller.go](#)

Purpose: The reconciliation engine that converts CodecoApp CRs into SWM Applications.

Key Functions:

- **Reconcile Loop:** Watches for CodecoApp resources and triggers reconciliation
- **CodecoApp → SWM Mapping:** Transforms CODECO application models to SWM Application and ApplicationGroup resources using the `MapToSWMApplicationModel()` function
- **Prometheus Integration:** Queries metrics for nodes, pods, and applications
- **Status Updates:** Populates CodecoAppStatus with live metrics every minute

Conversion Process (Apps to SWM Apps):

```
// MapToSWMApplicationModel copies CodecoApp spec to SWM Application
specfunc MapToSWMApplicationModel(codecoApp *codecov1alpha1.CodecoApp, swmApp *swmv1alpha1.Application) {
    copier.CopyWithOptions(&swmApp.Spec, &codecoApp.Spec, copier.Option{DeepCopy: true}) // Maps workloads, channels, and pod specs from CodecoApp to SWM format
}
```

The controller:

- Creates an ApplicationGroup named *acm-applicationgroup*
- Creates/updates an Application named *acm-swm-app* with workloads from the CodecoApp
- Maps channel settings (bandwidth, delay, framesize) to SWM network requirements

3.1.2.2 GUI Sub-component

Location: [gui/](#)

Structure:

- [gui/frontend/](#): React application with PatternFly UI
- [gui/backend/](#): Node.js API server

Features: rf. to Table 1

Table 1: Key features of the GUI sub-component

Feature	Description
YAML Generator	Form-based CodecoApp YAML creation
YAML Upload	Deploy configurations directly to Kubernetes
Resource Monitoring	View pods, namespaces, custom resources
CRD Management	List and inspect CRDs
Health Checks	Built-in health monitoring endpoints

API Endpoints:

- POST /apply-yaml - Deploy YAML to Kubernetes
- GET /get-status - Get resource status
- GET /api/namespaces - List namespaces
- GET /api/pods - List pods
- GET /crds - List CRDs



- GET /crds/microservices - List CodecoApp resources

3.1.2.3 Non-K8s Sub-component (codeco_nonk8s)

Location: [codeco_nonk8s/](#)

Purpose: Enables deployment of workloads to edge devices managed by Flightctl without requiring full Kubernetes on edge nodes.

Technology: Virtual Kubelet Provider

Architecture:

- **Direct Pod Pass-through:** Works with native v1.Pod objects
- **Flightctl Integration:** HTTP client for device and pod management
- **Virtual Node Representation:** Edge devices appear as virtual Kubernetes nodes

Project Structure:

```
codeco_nonk8s/
├── cmd/vk-flightctl-provider/    # Main entrypoint
├── pkg/
│   ├── provider/               # Virtual Kubelet provider
│   ├── flightctl/              # Flightctl API client
│   │   ├── client.go           # HTTP client
│   │   └── pods.go             # Pod management
│   └── models/                 # Data models
│       ├── device.go           # Edge device representation
│       ├── fleet.go            # Fleet grouping
│       ├── pod_mapping.go      # Pod-to-device tracking
│       └── target.go           # Device selection logic
```

Configuration:

- **ConfigMap vk-flightctl-config:** API URL and settings
- **Secret vk-flightctl-oauth:** Keycloak authentication credentials

3.1.2.4 OCM (Open Cluster Management) Policies

Location: [policies/](#)

Purpose: Responsible for the automated installation, configuration, and governance of the entire CODECO framework across the multi-cluster environment.

Key Functions:

- **Automated Framework Deployment:** Generates OCM Policies to ensure all CODECO components and dependencies (e.g., OCM, Prometheus) are installed in the correct locations.



Grant Agreement nr: 101092696

- **Enforcement:** Enforces adherence to configuration policies throughout the system's life, preventing erroneous or malicious modifications to the framework.
- **Onboarding:** Automatically configures and integrates new clusters as they are added to the federated environment.

Structure:

```

policies/
├── components/
│   ├── cert-manager/
│   │   └── manifests/
│   ├── kepler/
│   │   └── manifests/
│   ├── mdm-connectors/
│   │   └── manifests/
│   ├── multus/
│   │   └── manifests/
│   ├── nemesys-fc/
│   │   └── manifests/
│   ├── netma-nsm/
│   │   └── manifests/
│   ├── pdlc-ca/
│   │   └── manifests/
│   ├── pdlc-dp/
│   │   └── manifests/
│   ├── pdlc-gnn/
│   │   └── manifests/
│   ├── pdlc-integration/
│   │   └── manifests/
│   ├── pdlc-marl/
│   │   └── manifests/
│   ├── prometheus/
│   │   ├── manifests/
│   │   └── scripts/
│   ├── qos-scheduler/
│   │   ├── manifests/
│   │   └── manifestworks/
│   ├── secure-connectivity/
│   │   └── manifests/
│   └── workload-placement-solver/
│       └── manifests/
├── placement-rules/
├── scripts/
│   └── deploy-*.sh
├── ...
├── README.md
├── setup-prerequisites.sh
└── validate-deployment.sh

```

Cluster Labeling:

- *codeco-enabled=true*: Enable CODECO deployment
- *kepler-enabled=true*: Enable Kepler energy monitoring



Funded by
the European Union

- environment=production | staging | development

3.1.2.5 Neighbourhood Manager

Location: [controllers/neighbourhood_controller.go](#)

Purpose: Manages logical groupings of Managed Cluster resources—defined as "neighbourhoods"—associated with specific CODECO applications as proposed by the external CNC component.

Key Functions:

- **Metadata Propagation:** Uses OCM ManifestWorks to distribute neighbourhood definitions and cluster metadata to all constituent managed clusters.
- **Dynamic Lifecycle Management:** Continuously monitors for cluster events (deletions or metadata updates) and automatically synchronizes these changes across the federated environment.
- **Cluster Claims:** Utilizes custom ClusterClaim resources to reflect and reconcile updated cluster attributes on the Hub cluster.

3.1.2.6 Federated Monitoring (Prometheus & Thanos)

Location: [monitoring-architecture/prom_rules/](#)

Purpose: A tiered observability stack that scales from single-cluster metrics (Prometheus) to global, cross-cluster analysis (Thanos).

Components:

- **Prometheus:** Deployed in managed clusters to scrape local resources and evaluate recording rules for pod, service, and node-level metrics.
- **Thanos Receiver:** Provides a scalable remote write endpoint in the Hub cluster to ingest metrics from distributed Prometheus instances.
- **Thanos Querier:** Offers a unified, global query interface (PromQL) that aggregates data from all clusters for use by the ACM-FC and Grafana.
- **Thanos Ruler:** Evaluates recording and alerting rules at a global level across federated data sources.

3.1.3 Code Structure

```
acm/
├── api/v1alpha1/
│   └── codecoapp_types.go          # CRD Type Definitions
│                                   # CodecoApp, CodecoAppSpec,
│                                   CodecoAppStatus
├── bundle/                        # OLM Bundle for operator
├── deployment
├── codeco_nonk8s/                # Non-K8s/Edge deployment sub-
│   └── component
│       ├── cmd/vk-flightctl-provider/ # Virtual Kubelet main
│       ├── pkg/                      # Provider implementation
│       └── deploy/                   # Kubernetes manifests
├── config/
│   ├── cluster/                   # Kind cluster configurations
│   ├── crd/                       # CRD definitions
│   └── manager/                   # Controller manager deployment
```



rbac/	# RBAC configurations
samples/	# Sample CodecoApp YAMLs
olm/	# OLM subscription/catalog
controllers/	
codecoapp_controller.go	# Main reconciliation logic
helpersForSvmApps.go	# SWM Application helpers
gui/	
frontend/	# React/PatternFly UI
backend/	# Node.js API server
internal/qos-scheduler/	# Embedded SWM API types
api/v1alpha1/	# Application, Channel, NetworkPath
types	
monitoring-architecture/	# Prometheus/Thanos/Kepler setup
policies/	# OCM multi-cluster policies
prom_rules/	# Prometheus recording rules
scripts/	
post_deploy.sh	# Deploys all CODECO components
pre_deploy.sh	
after_netma_deployment.sh	
main.go	# Operator entrypoint
Makefile	# Build/deploy commands
Dockerfile	# Container image build

CodecoApp CRD Specification

API Version: codeco.he-codeco.eu/v1alpha1

Spec Fields: Rf. to Table 2-4. The fields follow the Golang parsing rules. Therefore, units are considered as in Golang.

Table 2: Main specification fields of the CodecoApp CRD

Field	Type	Description
appName	string	Application identifier
qosClass	Enum	Gold, Silver, Best effort
securityClass	Enum	High, Good, Medium, Low, None
compliance Class	Enum	High, Medium, Low
performanceProfile	Enum	Greenness, Resilience, UserDefined
appEnergyLimit	string	Max energy expenditure (percent)
appFailureTolerance	Enum	High, Medium, Low
codecoapp-msspec	array	Workload definitions

Table 3: Workload (microservice) fields defined in the CodecoApp specification



Field	Type	Description
serviceName	string	Microservice identifier
podspec	PodSpec	Kubernetes pod specification
serviceChannels	array	Communication channels

Table 4: Channel configuration fields and QoS parameters for service communication.

Field	Type	Description
channelName	string	Channel identifier
serviceClass	Enum	BESTEFFORT, ASSURED
otherService	ServiceId	Target workload connection
advancedChannel-Settings	object	QoS parameters
- Bandwidth	string	Minimum bandwidth in bits/second (e.g., "5M" for 5Mb/s)
- maxDelay	string	Maximum latency, e.g., "1s", "10ms" or "1000ns"
- frameSize	string	Bytes per frame
- sendInterval	string	Interval between frames in seconds

Status Fields (populated by controller):

- **status:** OK, Warning, Error
- **appMetrics:** CPU, memory, pod count aggregations
- **nodeMetrics:** Per-node resource consumption
- **serviceMetrics:** Per-service metrics with pod/node mapping

3.2 PDLC-FC: Privacy-preserving Decentralised Learning and Context-Awareness

3.2.1 Description

The CODECO PDLC component has been extended to support FC scenarios through targeted improvements in communication, knowledge exchange, and data preprocessing. The updates enhance inter-component communication, introduce federation-aware knowledge exchange mechanisms, and improve data normalization processes to ensure consistent handling and accurate aggregation across distributed environments.



PDLC provides distributed, privacy-preserving AI-based recommendations for workload management. Its architecture avoids single points of failure and supports dynamic and mobile execution environments, making it suitable for federated and multi-cluster deployments.

During the FC phase, the core architectural building blocks remain as described in Deliverable 13. Only incremental enhancements were introduced to improve operation in federated, multi-cluster setups. These refinements strengthen federation-aware communication, ensure robust knowledge exchange, and support scalable and consistent data normalization.

The main PDLC sub-components and their corresponding optimizations are.

- [Data Processing \(PDLC-DP\)](#) is responsible for the collection of Prometheus metrics derived from the NetMA, ACM, and MDM components, as well as for their normalization. At this stage, additional adjustments have been introduced to enable accurate normalization based on predefined metric intervals. Furthermore, optimizations have been applied to the integrated database to ensure data consistency and prevent data loss in the event of database instance failures.
- [Context Awareness \(PDLC-CA\)](#) is responsible for computing aggregate cost metrics for nodes and clusters, based on user-defined profiles such as greenness or resilience. This component has not undergone further optimizations in the FC phase, as its primary role remains the provision of aggregated cost indicators that support robust and informed recommendation processes.
- [PDLC Decentralised Learning based on Graph Neural Networks \(PDLC-DL GNNs\)](#) provides estimations of infrastructure utilization based on computational and networking metrics, generating node-level CPU and memory forecasts for 5, 15, and 30-minute horizons. In the FC phase, enhancements include the implementation of inter-GNN communication mechanisms that enable collaboration across federated clusters belonging to the same neighbourhood, thereby supporting decentralized and scalable inference.
- [PDLC Decentralised Learning based on Multi-Agent Reinforcement Learning \(PDLC-DL RL\)](#) provides intelligent application placement recommendations. In this phase, additional adjustments have been made to implement a communication protocol for knowledge exchange among distributed PDLC multi-agent reinforcement learning agents, enabling coordinated decision-making across federated clusters.

3.2.2 Sub-components

3.2.2.1 PDLC-DP

PDLC-DP component has been extended in the FC phase with several new functionalities that further support federated operation, represented in Figure 5. Particularly, PDLC-DP is responsible for retrieving the clusters' neighbourhood information by propagating the [Neighbourhood](#) configuration through a dedicated **ConfigMap**, which is subsequently made available to both the PDLC-DL (GNNs) and PDLC-DL (MARL) components. This enhancement enables federation-aware learning and coordination among clusters belonging to the same neighbourhood and requires aligned code contributions across all involved components to ensure consistent configuration handling.

Another key enhancement is the introduction of the *Synchronizator* module, which constitutes a new addition to PDLC-DP. The *Synchronizator* is responsible for ensuring data consistency across the data processing pipeline and for mitigating the risk of potential database instance failures. By maintaining synchronized and persistent data states, the module prevents scenarios in which a database instance loss could result in the loss of critical data required by the

remaining PDLC sub-components, thereby increasing the overall robustness and reliability of the system.

Regarding the [pdlc-synchronizer](#), the core functionality has been fully implemented to be part of the PDLC-DP component. However, it is not yet deployed in production mode. At the current stage, the module is considered to be at an advanced prototype level, having undergone initial functional verification to validate its design assumptions and operational logic.

Finally, PDLC-DP has refined the normalization process in order to provide more robust and reliable estimations, resulting in improved data quality for downstream processing. In particular, for metrics that require normalization, primarily compute and network-related metrics, PDLC-DP has been extended to additionally collect and manage metric value intervals, which are used to perform accurate and consistent normalization. The resulting normalized data are then forwarded through the InfluxDB instance to the remaining PDLC sub-components, ensuring that all learning and decision-making modules operate on harmonized and correctly scaled inputs.

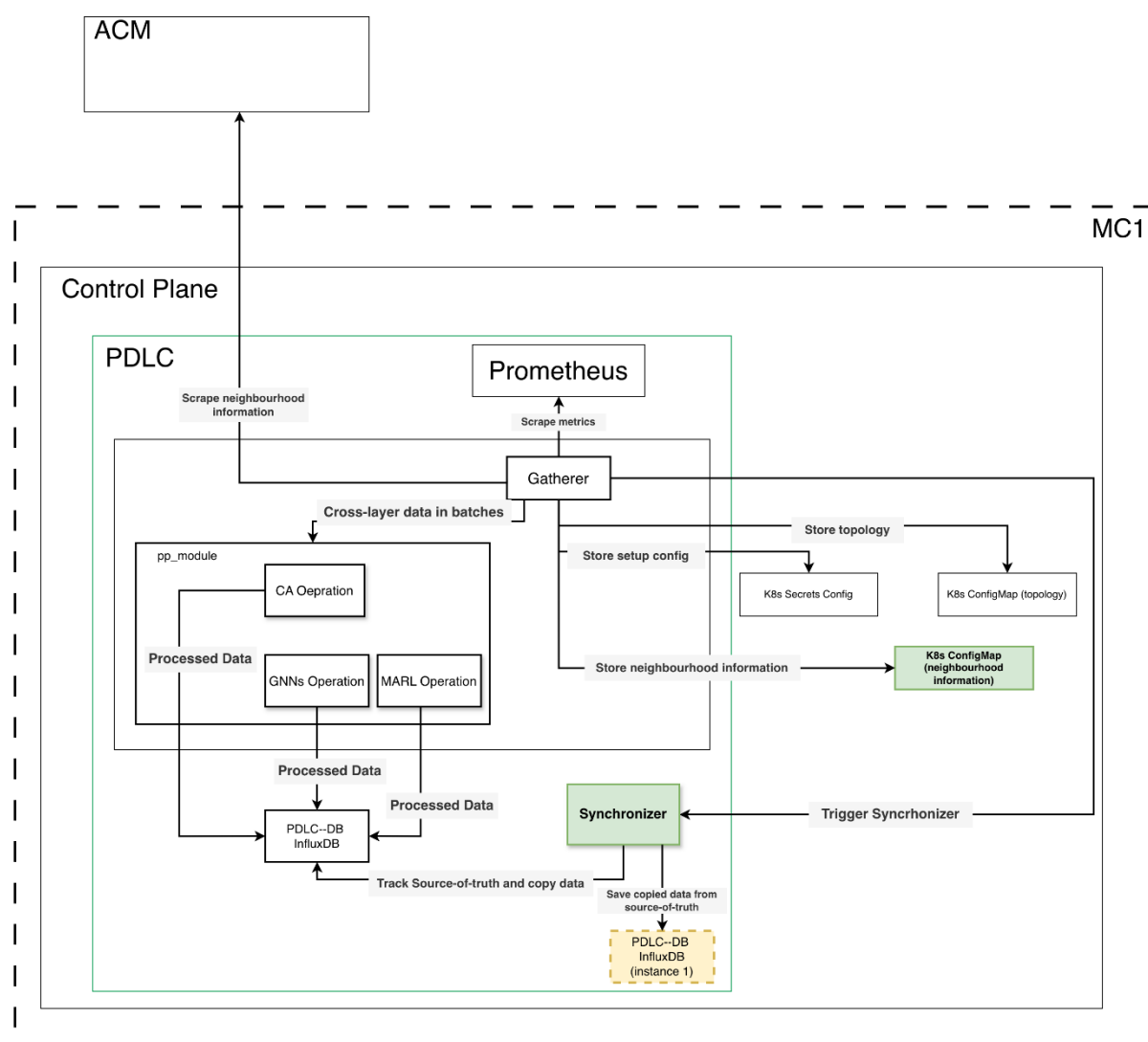


Figure 5: PDLC-DP functional scheme.

Figure 5 illustrates the functionality that the PDLC-DP follows. As reported, the PDLC-DP consists of two main functionalities: (a) the Gatherer and the (b) the pp module. The Gatherer is responsible for collecting cross-layer metrics from Prometheus, along with neighbourhood and

topology information, transmitting this data in batches, and normalizing it for further processing. The neighbourhood information refers to the information of nearby PDLC agents, which is required to establish and maintain communication among the participating clusters within a given neighbourhood. By leveraging this information, the **PDLC-DL** sub-component is able to implement collaborative learning mechanisms across clusters, which constitute a core functionality of the PDLC. The **pp_module** performs the core data processing functions of the PDLC-DP by applying CA, GNNs, and MARL pre-processing operations to the collected data, generating processed outputs that are stored in the PDLC-DB (InfluxDB). Through these two functionalities, the PDLC-DP enables consistent data collection, real-time processing, and reliable persistence and replication of control-plane data. A new functionality refers to the Synchronizer, which enables the replication of the data that has persisted in the PDLC-DB in case that it fails to unpredictable conditions.

Specifically, the **Gatherer** is responsible for triggering the **Synchronizer**, designating the core **PDLC-DB** instance as the source of truth, and enabling real-time replication of the stored data to a secondary PDLC-DB instance to ensure continuity in case the core instance fails due to unpredictable conditions. This functionality is essential for preserving the data required by the PDLC to execute its complete workflow and for preventing interruptions to the overall CODECO workflow execution. The **Synchronizer** is also responsible for creating additional PDLC-DB instances when the active replica fails and for reassigning the source-of-truth label to the appropriate instance, thereby ensuring uninterrupted data replication and system operation.

3.2.2.1.1 Implementation details

PDLC-DP was enhanced by extending existing mechanisms rather than introducing major architectural changes. One key improvement is enabling communication between PDLC agents within the same neighbourhood to support collaborative learning. Since the PDLC-DP Gatherer already retrieves topology data and InfluxDB credentials, its role was extended to also fetch neighbourhood information from the ACM Neighbourhood CRD. This information is required by the PDLC-DL component to perform knowledge exchange while preserving privacy and security.

The Gatherer runs continuously, monitors the Neighbourhood CRD, retrieves updated neighbourhood data, and stores it in a Kubernetes ConfigMap consumed by PDLC-DL. When mobility events occur (e.g., a new cluster joins the neighbourhood), the Gatherer automatically updates the ConfigMap so that PDLC-DL always operates with the latest neighbourhood context.

PDLC-DP also improves real-time data normalization. The *pp_module*, responsible for pre-processing, now uses predefined domain-specific normalization intervals instead of relying only on dynamic min/max values. Dynamic scaling can be inaccurate when limited historical data is available. Using realistic per-metric intervals ensures stable and consistent normalization during online operation. This improves the reliability of the normalized metrics consumed by downstream components, including PDLC-CA and PDLC-DL.

Another important addition is the Synchronizer, which ensures data availability and integrity for AI-driven orchestration. The Synchronizer continuously replicates data from the primary PDLC-DB (InfluxDB), which acts as the source of truth, to a secondary database instance. If the primary instance fails, the secondary is promoted to source of truth to maintain uninterrupted operation. The Synchronizer monitors the health of the PDLC-DB in real time. If a failure is detected, it starts a new database instance, generates new credentials, and makes them available to the other PDLC components. Replication then resumes to ensure continuous data availability and prevent data loss across the PDLC workflow.

3.2.2.1.2 Code structure



The PDLC-DP sub-component main source code files are briefly summarized in Table 5:

Table 5: PDLC-DP sub-component source code

File	Function
Neighbourhood retrieval	
pdlc-dp/src/k8s_calling/k8s_class.py	The newly introduced logic focuses on reading Neighbourhood CR data in order to capture the set of managed clusters. Specifically, it retrieves the first available Neighbourhood CR, extracts the names of the clusters defined in its managed Clusters specification, and stores this information in a dedicated ConfigMap for subsequent use by other PDLC subcomponents.
Synchronizer	
pdlc-synchronizator/src/main.py	A script that monitors the health logs of the PDLC-DB (InfluxDB) instance, designates the active instance as the <i>source of truth</i> , and, upon detecting an instance failure, provisions a new PDLC-DB instance, promotes it to the new source of truth, and replicates the data from the previous instance.
Optimized Normalization	
pdlc-dp/src/pp_module/utilities.py	Optimized script that applies domain-specific value intervals to ensure accurate and stable metrics normalization.

3.2.2.1.3 Selected Technologies

The PDLC-DP sub-component is fully designed and implemented using the Python programming language (version 3.10 or higher). In this version, no additional third-party libraries or dependencies have been introduced, the implementation continues to rely exclusively on the existing, previously integrated Python packages.

3.2.2.1.4 Pre-requisites

Refer to the [README.md](#) of PDLC-DP code for a detailed list of requirements:

- At least a Hub and an MC need to be set.
- CODECO-FC and respective requirements need to be installed.
- Prometheus monitoring tool for scraping the cross-layer and custom metrics derived by the ACM-FC Agent, MDM-FC Connectors, and NetMA-FC components.
- The PDLC-DP components (Gatherer and Preprocessing modules) are to be installed as a standalone service. During this phase of the project, the PDLC-DP also initializes the PDLC-DB component (InfluxDB). Once initialized, the PDLC-DP expects to receive metrics from the Prometheus instance located in the respective MC, to collect cross-layer and custom metrics.

3.2.2.1.5 Installation Guide

A detailed installation guide is available with the code of [PDLC-DP \(readme\)](#).



3.2.2.1.6 Inputs and Outputs

In this version of the PDLC-DP, the input and output interfaces remain unchanged. The component continues to collect metrics from the Prometheus instance, which are exposed by the ACM-FC Agent, MDM-FC connectors, and the NetMA-FC of the respective MC. These inputs are processed to produce pre-processed metrics tailored to the specific requirements of the PDLC sub-components (PDLC-CA, PDLC-DL-GNNs, and PDLC-DL-MARL). The resulting datasets are stored in the PDLC-DB (InfluxDB) using the corresponding buckets and measurements.

The only functional enhancement introduced in this version concerns the Gatherer PDLC-DP sub-component, which has been extended to additionally collect neighbourhood CR information.

3.2.2.2 PDLC-CA: Context-aware Cost Estimation Module

[PDLC-CA](#) is a sub-module of PDLC responsible for computing **context-aware cost perspectives** for nodes and clusters during the deployment phase. Its primary role is to translate application-level intent (as expressed in the CAM) into quantitative cost indicators that can be consumed by the scheduling layer.

At deployment time, PDLC-CA evaluates candidate nodes and MCs within the selected neighbourhood and produces annotated cost values aligned with the target performance profile defined in CAM.

In the current CODECO-FC integration, PDLC-CA consumes multi-dimensional inputs collected through the CODECO observability pipeline, including:

- **Compute metrics** (CPU utilization, memory availability, energy state)
- **Network metrics** (latency, bandwidth, link energy, flow energy, path characteristics)
- **Data metrics** (data locality, compliance constraints, data size)
- Additional infrastructure metrics exposed via Prometheus

These inputs are normalized and combined into composite cost indicators. The aggregation logic depends on the selected performance profile, such as:

- Greenness (energy-aware optimization)
- Resilience
- Latency-sensitive performance
- Other user-defined target profiles expressed in CAM

The performance profile is selected by DEV in the CAM, allowing the developer to explicitly select optimization priorities at deployment time.

Importantly, PDLC-CA does not execute placement decisions. It computes cost perspectives and writes them to the appropriate CRDs (e.g., as annotated node costs in the AssignmentPlan), where they can be consumed by SWM-FC for policy-filtered scheduling decisions. Currently, the composite costs generated by PDLC-CA are used by PDLC-RL to generate recommendations to PDLC-MARL.

For a deeper explanation of the computational principles and optimization rationale behind PDLC-CA (in single-cluster mode), refer to Deliverable D31.

3.2.2.2.1 Sequence Diagram and Operational Workflow



The PDLC-CA component is responsible for computing context-aware cost metrics that guide workload placement across clusters in the CODECO architecture. Figure 6 provides the overall operational workflow. The process is implemented through a series of Python scripts that interact with external systems such as InfluxDB and the Kubernetes API. The workflow is orchestrated by the `main.py` script, which acts as the central controller for the pre-processing and evaluation pipeline.

The process begins with `main.py` invoking `influx_handler.py` to retrieve the most recent node-level metrics from InfluxDB. These metrics are stored in the `context_metrics` measurement and typically include values such as network delay, energy usage, and compliance indicators. Once retrieved, the data is passed back to `main.py`, which then delegates to `cost_evaluator.py` to fetch the cost evaluation profiles. These profiles define how different metrics should be weighted and normalized and are stored in a Kubernetes ConfigMap. `cost_evaluator.py` accesses the Kubernetes API to read this configuration and returns the necessary profile data to `main.py`.

With both raw metrics and cost profiles available, `main.py` calls `process_and_store_costs()` in `cost_evaluator.py`, which performs the core computation. This involves applying predefined cost equations to the input data to derive node- and cluster-level scores. These scores represent how suitable each node is for hosting workloads, considering the current context.

Finally, the computed scores are passed back to `influx_handler.py`, which writes them to the InfluxDB time-series database under a dedicated PDLC-RL bucket. This makes the data available for downstream consumers, such as PDLC-DL or even the SWM scheduler or other CODECO orchestration components, enabling them to make informed placement decisions based on real-time, context-aware information.

Overall, the PDLC-CA component operates as a modular, stateless evaluator that transforms raw system metrics into actionable cost profiles used throughout the federated CODECO platform.

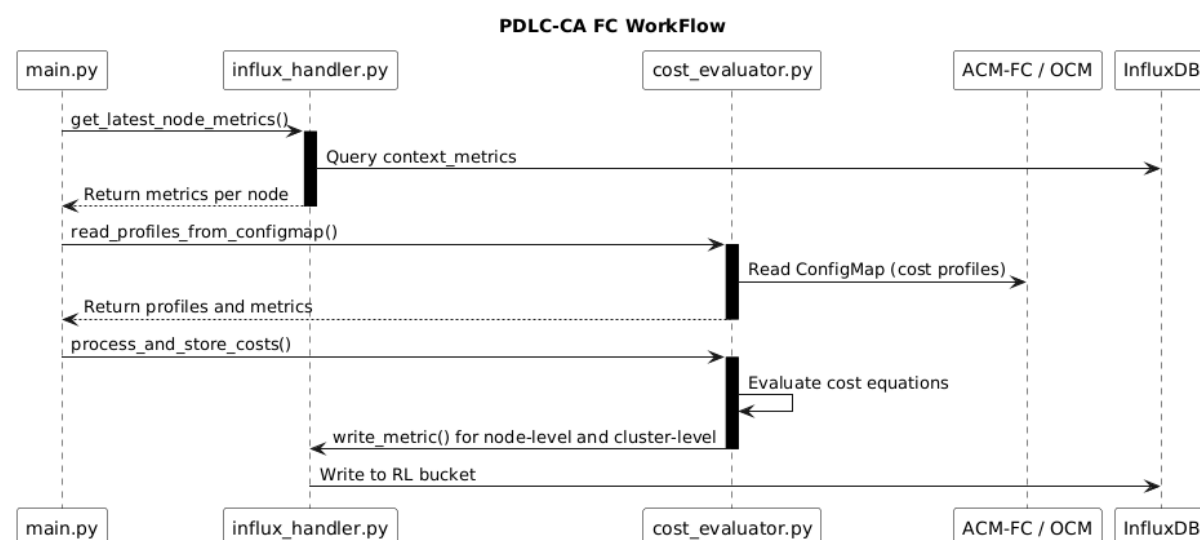


Figure 6: PDLC-CA workflow, FC operation.

3.2.2.2.2 Implementation details

The PDLC-CA-FC implementation consists of a modular Python-based pipeline that retrieves context metrics, evaluates node and cluster cost profiles, and stores results for orchestration decisions in the CODECO framework. The PDLC-CA-FC branch provides a clean, modular

implementation of the context-awareness engine for federated CODECO environments. It separates concerns between:

- Control logic (main.py)
- Data access (influx_handler.py)
- Cost modelling (cost_evaluator.py)

The **configuration setup** is based on the [pdlc_ca_configmap.yaml](#) ConfigMap, which can be used by user DEV to define metrics to use (e.g., delay, energy), weights, normalization rules. It can also be used to consider scoring-based on user-defined metrics.

3.2.2.2.3 Pre-defined Target Performance Profile Scoring: Resilience and Greenness

PDLC-CA-FC provides a way to rely on the CAM defined target performance profiles, and to consider new profiles. The pre-defined profiles are resilience and greenness. The way the aggregated score to define resilience or greenness is provided can be changed by the user, by providing new equations based on available, normalized CODECO metrics.

An example for a specific aggregated cost profile for node scoring (resilience) is provided in Figure 7 and for greenness (where the queries were defined by the user) is provided in Figure 8. In this example CO2 is assumed to be custom-defined metrics, and *avgNodeEnergy* is a CODECO pre-defined metric, collected by ACM-FC. Their normalized forms are referenced with the *normalized_* prefix.

```
profiles:
- name: Resilience
  equation: "normalized_avgNodeFailure * normalized_uNodeNetFailure * 1 / oNodeDegree"
```

Figure 7: PDLC-CA-FC, example for cost scoring defined for node resilience.

```
profiles:
- name: UserDefined
  equation: "cm_normalized_co2 * normalized_avgNodeEnergy" # Mixes custom and built-in metrics
```

Figure 8: Pre-defined target profile greenness, with equation defined by the user, based on CO2 foot printing.

3.2.2.2.4 User-defined Node and Cluster Scores

In addition to the pre-defined target profiles and respective heuristics, a user can define additional target profiles and change the scoring functions applied to pre-defined target performance profiles. To ensure consistent behaviour, only specific metrics are **available in normalized form** and can be safely used in profile cost equations. In the current code version, metrics available that start with the prefix *normalized_* can be used in profile equations.

Examples of normalized metrics that can be used to generate user-defined aggregate costs for nodes and cluster:

- *normalized_avgAppCpu*
- *normalized_avgAppMemory*
- *normalized_avgNodeEnergy*
- *normalized_avgNodeFailure*
- *normalized_oNodeNetFailure*

- normalized_uNodeNetFailure
- normalized_uNodeBandWidth
- normalized_uLinkEnergy
- uNodeDegree # ****not normalized****

Note:

1. Do **not** use the reserved profile names `Greenness` and `Resilience`, as they are predefined examples.
2. When creating **custom metrics**, avoid reusing any metric names or equations defined in Annex I.

An example for a defined custom target profile is provided in Figure 9. If developers want to define custom metrics to be used in the aggregated score equations along with their Prometheus queries and normalization ranges, they must follow the format provided in Figure 9. Specifically:

1. **name:** Logical name of the metric (referenced as `cm_normalized_<custom_metric_name>` in profile equations)
2. **promQL:** The raw Prometheus query for the metric must include **instance label**, which identifies the Kubernetes **node name**
3. **min, max:** Required to enable proper normalization of the raw metric value

```
custom_metrics:
# CO2 footprint metric (example; requires exporter pushing node_co2{instance} to Prometheus)
- name: co2
  min: 0
  max: 20000
  promQL: 'avg by (instance)(irate(node_co2{instance}[5m]))'
```

Figure 9: PDLC-CA-FC example for defining a custom target performance profile.

3.2.2.2.5 Implemented Strategies for Cluster Scoring

In the context of federated application placement (e.g., within CODECO), cluster-level ranking is required to evaluate how well a cluster meets a desired performance profile.

In the current PDLC-CA-FC implementation, two strategies are considered to provide cluster level scoring based on node scores:

- **Strategy 1: average of node scores.** This method computes the mean score across all nodes in the cluster. It assumes that all nodes contribute equally to the overall performance of the cluster. The cluster score reflects the overall suitability, smoothing out variations across nodes. It is appropriate when the application workload needs to be distributed across multiple available nodes, and emphasizes the average performance scoring, over extremes (min, max). It is more suited for general purpose application deployment.
- **Strategy 2: Norm of score vector.** This strategy treats the node scores as a vector in N-dimensional space and computes its Euclidean norm (L2 norm). It captures both the magnitude and variance of the individual node scores. Clusters with many high-performing nodes will naturally rank higher. It emphasizes the aggregate strength of the most “popular

nodes" (in terms of nodes in the cluster), penalizing low-performing nodes. It is therefore considered suitable for resource-intensive or bursty application deployment.

3.2.2.2.6 Code Structure

The PDLC code structure is represented in Table 6.

Table 6: Summary of main PDLC-CA source code files.

File	Function
main.py	The entry point script that coordinates the entire CA processing loop.
influx_handler.py	Manages InfluxDB connections
	Data extractor, it fetches the normalized metrics from the data_CA.csv file stored in the PDLC shared volume and makes the metrics available as a return of the function <code>get_metrics()</code> .
cost_evaluator.py	Calculates normalized cost profiles
pdlc-ca-configmap.yaml	Enables user defined cost computation rules (weights, metrics)
ca-controller-deployment.yaml	Deploys the module as a pod in the Hub cluster

3.2.2.2.7 Installation Guide

A detailed installation guide is available with the code of [PDLC-CA \(readme\)](#).

3.2.2.2.8 Selected Technologies

PDLC-CA has been implemented in Python 3 Input and output are based on YAML (CRD).

3.2.2.2.9 Pre-requisites

Refer to the [README](#) of PDLC-CA code for a detailed list of requirements:

- At least a Hub and an MC need to be set.
- CODECO-FC and respective requirements need to be installed.
- InfluxDB (PDLC-DB) MUST be accessible
- Prometheus (for metrics)
- Kubernetes ConfigMap: `performance-profiles`
- Kubernetes Secret: `influxdb-token`
- InfluxDB buckets:
- CA: for reading input metrics # Generated by PDLC-DP
- RL: for writing evaluated costs

3.2.2.2.10 Inputs and Outputs

1. Input of PDLC-CA corresponds to CODECO metrics and user-defined custom metrics, both normalized from, provided via PDLC-DP in InfluxDB.
2. Output of PDLC-CA is provided in InfluxDB.



3.2.2.3 PDLC-DL – FedGNN

The [PDLC-DL-GNNs](#) sub-component consists of three modules represented in Figure 10: the **Decentralized Federated Training** pipeline, the **Controller**, and the **Inference API**. The STGNN model produces per-node CPU and memory utilization forecasts at 5-, 15-, and 30-minute horizons, which are written to dedicated measurements in the RL InfluxDB bucket and read by the PDLC-MARL module as input to its placement decisions.

At the current stage, PDLC-DL-GNN can be used, but is not an integrated part of PDLC.

The Controller runs on a one-minute cycle, querying the *cpu_memory_metrics* measurement from the GNN InfluxDB bucket — data written there by the PDLC-DP subcomponent. On each cycle, it constructs a request payload containing the latest per-node observations and current cluster topology, and issues an HTTP call to the Inference API. The Inference API resolves and loads the appropriate pre-trained model from the Model Registry, selecting it based on the number of active nodes, the target metric, and the requested forecast horizon. The resulting forecasts are written back to InfluxDB under *forecasts5*, *forecasts15*, and *forecasts30*.

The Decentralized Federated Training pipeline supplies the Model Registry with updated model weights. Each cluster trains its STGNN locally and exchanges only encrypted, masked hidden-layer weights with neighbouring clusters via peer-to-peer API calls, without sharing raw monitoring data. As illustrated in Figure 10, model update orchestration and weight storage are handled centrally within each cluster to maintain consistency across federated training rounds.

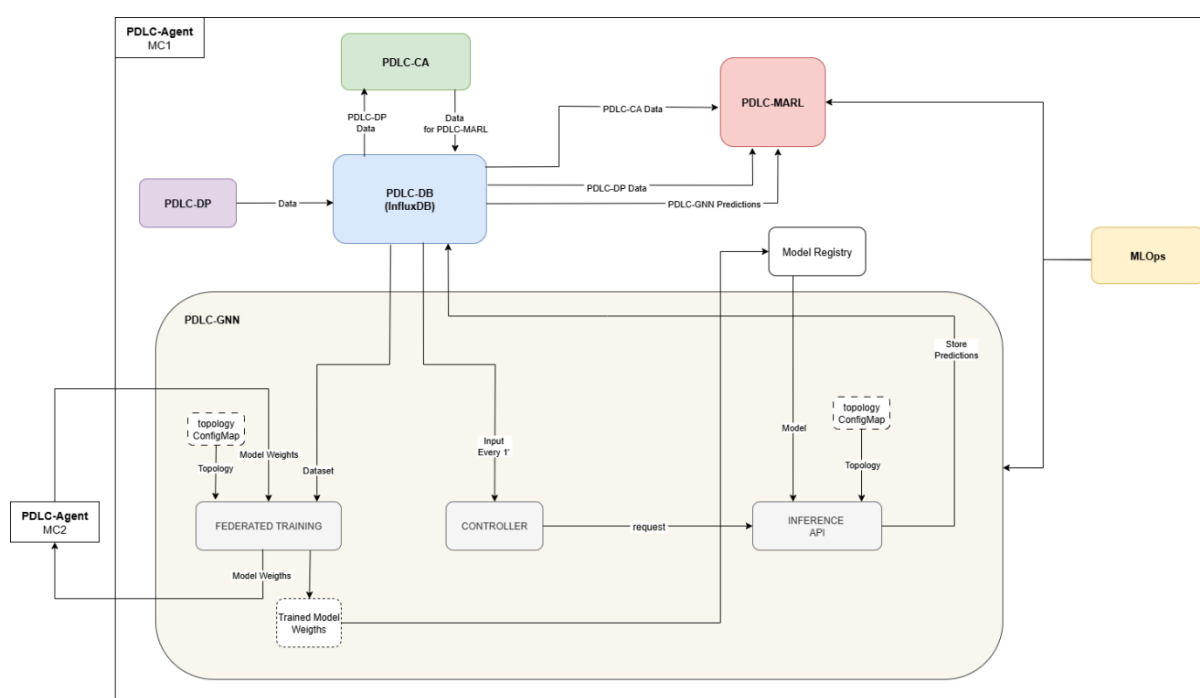


Figure 10: PDLC-DL-GNNs functional scheme.

3.2.2.3.1 Sequence diagram and Operational Workflow

The workflow associated with the PDLC-DL-GNN subcomponent in the FC phase, is illustrated in Figure 11 and involves continuous coordination between the GNN Controller and the Inference service. The GNN Controller is responsible for periodically retrieving monitoring data, specifically CPU and memory metrics, from the “GNN” InfluxDB bucket. These metrics, written by the PDLC-DP subcomponent under the “*cpu_memory_metrics*” measurement, are fetched every minute for all nodes in the local cluster. Based on this data, the Controller constructs an

appropriate request body and sends it as an API call to the Inference Service every minute. On the receiving end, the Inference Service continuously monitors the current cluster topology. Upon receiving an inference request, it identifies and loads the appropriate pre-trained GNN model from the Model Registry. Model selection is based on several factors: (i) the number of active nodes in the cluster, (ii) the target metric (CPU or memory), and (iii) the requested forecast horizon (5, 15, or 30 minutes). Once the Inference service is executed, the resulting forecasts are stored in the “*RL*” InfluxDB bucket under the corresponding measurements: “*forecasts5*”, “*forecasts15*”, and “*forecasts30*”, respectively. The pre-trained models are stored in the Model Registry component. The models are dependent on the number of nodes and, therefore, for the OSS CODECO Federated Toolkit, the existing pre-trained models currently correspond to 2 up to 8 node cluster topologies. Within the Federated Cluster (FC) phase, model training adopts a decentralized federated learning approach, enabling peer-to-peer training without raw data exchange. Each cluster performs locally training of the GNN model with respect to its own private data. During intermediate stages of training, each cluster exchanges masked model parameters (weights) with its immediate neighbours. These weights are securely aggregated using a privacy-preserving scheme, and local training proceeds with updated parameters. This train-and-aggregate cycle is repeated iteratively until certain conditions are met.



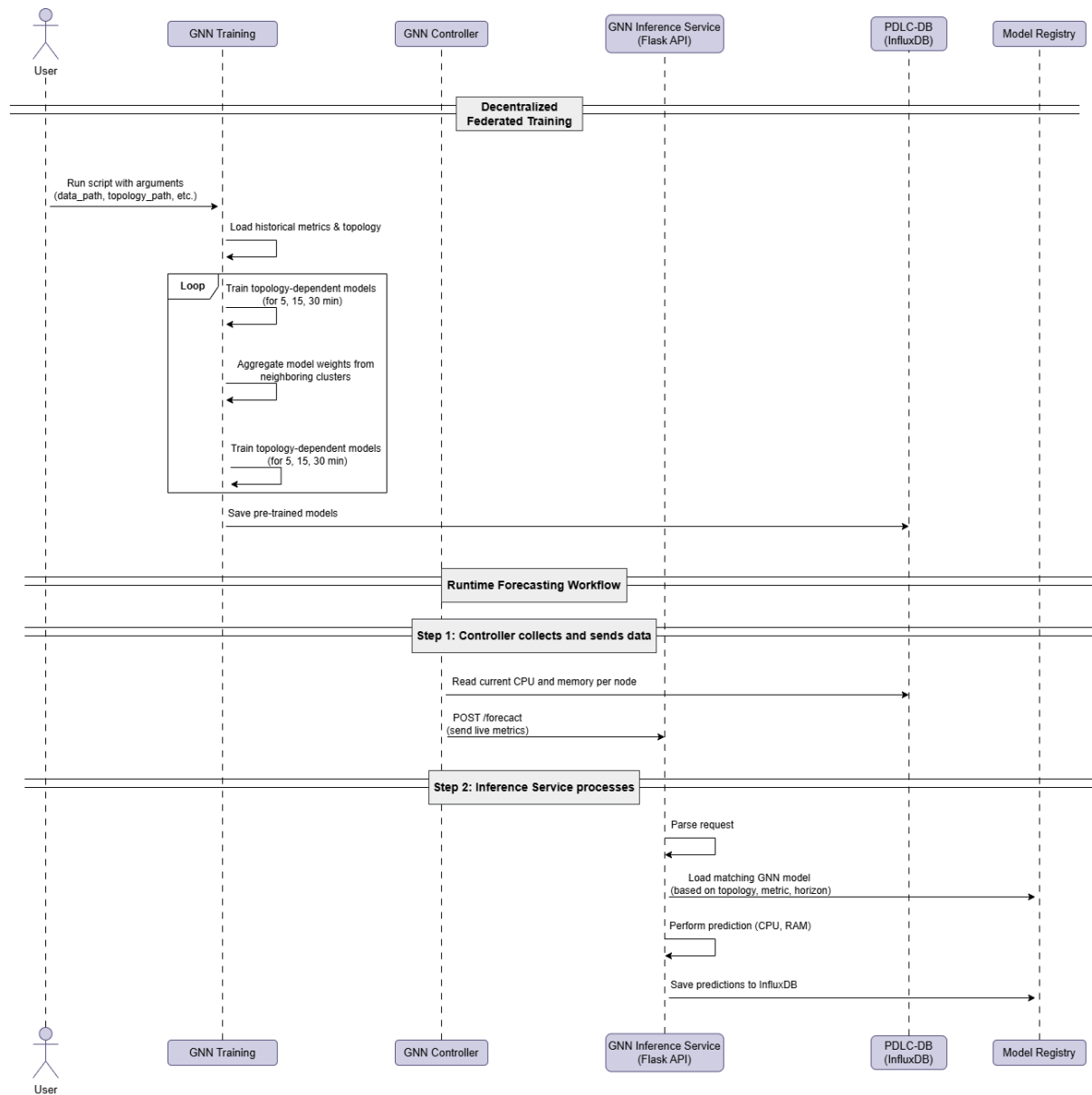


Figure 11: PDLC-DL-GNNs sequence diagram.

3.2.2.3.2 Component description with new features and extensions

In the FC phase, the PDLC-DL-GNN subcomponent extends its SC phase functionality with Decentralized Federated Training support. Each cluster trains its STGNN independently using data from its local GNN InfluxDB bucket, and exchanges only masked hidden-layer weights with immediate neighbours via POST /upload-weights. Raw monitoring data never leaves the cluster.

Each cluster maintains its own model instance in the local Model Registry. Weight updates are propagated peer-to-peer, meaning each cluster communicates only with its direct neighbours. Training rounds execute in parallel across clusters, and aggregation is performed locally via `aggregate_hidden_weights()` at the end of each round.

When a node rejoins the cluster, the Inference API resolves the appropriate pre-trained model from the Model Registry based on the current node count. If no matching model exists, the MLOps component triggers re-training. The same re-training trigger applies when a monitored performance drop is detected on the active model.

Model versioning is maintained in the Model Registry. If an aggregated weight update causes inference instability or propagates degraded behaviour, the system rolls back to the last stable model version. Online federated re-training is supported to handle gradual workload drift without requiring a full retraining cycle from scratch.

3.2.2.3.3 Implementation details

The FC phase introduces two primary implementation changes to the PDLC-DL-GNNs framework: the migration from PVC-based storage to InfluxDB, and the addition of decentralized federated training support.

Data storage and retrieval across the training, Controller, and Inference modules is handled via dedicated Python scripts using the influxdb-client library. The training and Controller modules query the GNN bucket under the `cpu_memory_metrics` measurement to retrieve per-node CPU and memory time-series data. The Inference module writes forecast outputs to the RL bucket under `forecasts5`, `forecasts15`, and `forecasts30`.

3.2.2.3.4 Model training procedure

The STGNN model is implemented via the LSTMGC class in `training.py`, combining a GraphConv layer for spatial feature extraction, an LSTM layer for per-node temporal modelling, and a Dense output layer for multi-step forecasting. The GCN supports configurable aggregation strategies (e.g., mean, sum), and cluster topology is encoded via the Graph Info class.

Model Structure and Training

Each cluster independently trains a model using a hybrid architecture consisting of:

- A **Graph Convolutional Layer (GraphConv)** for spatial feature extraction across nodes.
- A **LSTM Layer** for temporal modelling of time-series data per node.
- A **Dense Layer** to output multi-step forecasts.

This is implemented via the LSTMGC class in `training.py`, which combines the GCN and LSTM in a single module. The GCN uses a flexible aggregation strategy (e.g., mean, sum), and neighbour topology is encoded via the GraphInfo class.

Hidden Weight Extraction and Aggregation

Only trainable hidden-layer weights are extracted and shared, using three functions:

- `get_hidden_weights(model)` : extracts GraphConv and LSTM weights.
- `aggregate_hidden_weights(weights_list)` : averages each layer type independently across all cluster models.
- `set_hidden_weights(model, aggregated)` : apply the new aggregated weights to a local model.

This minimizes data transfer size and enforces privacy by avoiding exposure of model-specific output layers or embeddings.

Secure Weight Masking with Diffie-Hellman and PRG

A major extension of our decentralized pipeline is the integration of secure weight exchange using encryption. Before sharing any model weights, each cluster applies a masking scheme that prevents the exposure of raw weights, even during transmission and aggregation. This is done using the following components:



- **Diffie-Hellman Key Exchange.** Each cluster independently generates a private/public key pair using `generate_dh_keypair(p,g)` where p is a large prime (shared among all participants) and g is a primitive root modulo p . Clusters then exchange public keys and compute pairwise shared keys via `derive_shared_key(private_key, peer_public_key, p)`.
- **Pseudorandom Mask Generation (PRG).** Using the derived shared keys, a deterministic pseudorandom generator (PRG) creates noise tensors of the same shape as the model weights: `prg(shared_key, shape, seed)`. This ensures consistent noise generation between any two clusters using the same shared key and seed.
- **Asymmetric Noise Masking.** The masking function `mask_model_weights(weights_dict, shared_key_row, cluster_id, n_clusters)` adds or subtracts PRG-generated noise to/from each weight tensor. The noise is asymmetric and pairwise balanced, this design ensures that during aggregation, the total noise cancels out. Only the original average of the weight remains, while individual model weights are protected during transmission.

Federated Training Orchestration

The federated training orchestrates the following:

- **Local Model Training.** Each cluster trains its own model independently on local data. Once training is complete, the model weights are extracted and securely masked as described above.
- **Model Weight Collection via API.** Instead of accessing model files via shared volume mounts, they are collected via a Flask-based APIs (`STGNN_api/`), issued as HTTP requests. Each peer cluster exposes a `/upload-weights` endpoint that accepts zipped masked model weights.
- **Aggregation.** This collected hidden weights are aggregated using a layer-wise averaging strategy. This results in a new set of shared weights representing the consensus across all clients.
- **Redistribution.** The aggregated weights are injected into the local STGNN model instance. The updated model is saved and can be redeployed for inference.

Model Exchange via API

Each cluster hosts a Flask server that exposes:

POST `/upload-weights` accepts compressed model weight files (e.g., `trained_stgnn_model_cpu_weights.zip`) and unpacks them locally.

Example exchange workflow:

Cluster A finishes training, encrypts and masks its weights, and sends them via API to Cluster B. Cluster B receives multiple peer weights (A, C, D ...), aggregates all received weights with its own, and updates its local model. This process can be repeated for multiple rounds.

3.2.2.3.5 Code structure

The PDLC-DL-GNN sub-component main source code files are briefly summarized in Table 7.

Table 7: PDLC-DL-GNN sub-components source code

File	Function
Controller	
timeseries_data.py	Module that retrieves data from InfluxDB every minute and stores it locally.
Inference	
app.py	Code that implements the Inference Service API, detects the number of nodes in the topology dynamically, selects the appropriate models based on the number of nodes, makes the predictions, writes the forecasted values in the appropriate files and stores them into InfluxDB.
Training	
download_dataset.py	Code that downloads data from InfluxDB to be used for training.
Federated Training	
training.py	Code that defines the STGNN model architecture, including custom GraphConv and LSTMGC layers. Implements core utilities for federated learning, including weights extraction, aggregation, and secure encryption protocols based on Diffie-Hellman key exchange and pseudorandom masking.
main.py	Code that aggregates the hidden layers masked weights (GraphConv and LSTM) from several pre-trained STGNN models trained on different topologies and saves the aggregated model.

3.2.2.3.6 Installation Guide

To deploy and test the GNNs subcomponent:

1. Deploy the Inference API Server in the he-codeco-pdlc namespace

The [gnn_inference.yaml](#), specifies the environment variables of pod.

The deployment is performed with the following commands:

```
cd STGN_controller/inference
kubectl apply -f gnn_inference.yaml -n he-codeco-pdlc
```

This file includes both **Deployment** and **Service** specifications. Once applied, a pod, a deployment, and a service are created in the K8s cluster.

Docker image: hecodeco/pdlc-dl-gnns-stggn-inference:3.0.0 (available on Docker Hub)

2. Deploy the Orchestrator

The [gnn_controller.yaml](#), specifies the environment variables of pod.

The deployment is performed with the following commands:



```
cd STGNN_controller/orchestrator
kubectl apply -f gnn_controller.yaml -n he-codeco-pdlc
```

This file includes a **Deployment** specification. Once applied, a pod, and a deployment are created in the K8s cluster.

Docker image: hecodeco/pdlc-dl-gnns-stgnn-controller:3.0.0 (available on Docker Hub)

3. Deploy the Training Pod

The [gnn_training.yaml](#), specifies the environment variables of pod.

The deployment is performed with the following commands:

```
cd STGNN_training
kubectl apply -f gnn_training.yaml -n he-codeco-pdlc
```

This file deploys a **training pod** for running GNN model training tasks in the cluster.

Docker image: hecodeco/pdlc-dl-gnns-stgnn-training:3.0.0 (available on Docker Hub)

4. Deploy the Federated Training Pod

The [gnn_federated_training.yaml](#), specifies the environment variables of pod.

The deployment is performed with the following commands:

```
cd STGNN_federated_training
kubectl apply -f gnn_federated_training.yaml -n he-codeco-pdlc
```

This file deploys a **federated training pod**, for federated training across clusters.

Docker image: hecodeco/pdlc-dl-gnns-stgnn-federated-training:3.0.0 (available on Docker Hub).

3.2.2.3.7 Selected Technologies

- **Python:** programming language, version 3.10+.
- **InfluxDB:** an open-source time series database designed for high-performance storage, querying, and analysis of time-series data.

3.2.2.3.8 Pre-requisites

- At least a Hub and a MC need to be set.
- CODECO-FC and respective requirements need to be installed.
- In terms of data the PDLC-DB (InfluxDB) component is required to retrieve the input monitoring data and to store the output forecasts.
- The Cluster Topology ConfigMap (to identify the set of nodes and their relationships within each cluster) and the Neighbourhood Topology ConfigMap (to identify the set of clusters and their relationships within each neighbourhood) need to be available.
- The pre-trained models must exist in the Model Registry component.

3.2.2.3.9 Inputs and Outputs

The input data of the GNN inference service is retrieved from the “GNN” InfluxDB bucket and specifically the “*cpu_memory_metrics*” measurement. It includes the timestamp of the given values, the names of each node, the normalized CPU and memory values, as well as the minimum and maximum values with which the normalized values were calculated for both CPU and memory.

The output of the GNN inference service is stored in the “RL” InfluxDB bucket, using three dedicated measurements: `forecasts5`, `forecasts15`, and `forecasts30`, corresponding to forecast horizons of 5, 15, and 30 minutes, respectively. Each entry in these measurements includes the forecast timestamp, the name of each node, and the denormalized CPU and memory forecast values. In addition to this, the same forecast data is also written to local files located in the ‘outputs’ directory, specifically `forecasts5.csv`, `forecasts15.csv`, and `forecasts30.csv`, these files serve as auxiliary outputs for debugging, traceability, or offline analysis purposes.

3.2.2.4 PDLC-FC-DL-MARL

3.2.2.4.1 Sequence Diagram and Operational Workflow

The [PDLC-MARL-FC](#) subcomponent focuses on leveraging data in real time in order to provide dynamic and resilient recommendations for incoming application workloads. The main focus of the design of this subcomponent was to work on previously known weak points of the latest version of PDLC-RL, more specifically:

- **Quality and availability of data used:** Relying on static, file-based data collection hindered real-time processing speeds, which negatively impacted model performance.
- **Resilience of the solution:** In the first phase of CODECO, the RL component had a high dependency on infrastructure, meaning that if the number of nodes in a cluster were to change, or if applications stopped working, it did not have any way of reacting, thus decreasing the overall resilience of the system.
- **Adaptive solutions over time:** A major limitation was the use of static models that did not learn over time. This prevented “on-premises” fine-tuning and kept the system from evolving with the environment.
- **Non-distributed in between clusters:** The solution was restricted to single-cluster operations, lacking the distributed capabilities needed for broader usability across multiple environments.

Because of all of this, we decided to fully redesign part of the RL architecture to fit these key requirements. Since the integration of these changes the component has remained similar in terms of code implementation and architecture when compared to the deliverable D13. The focus during the last months of the project has been into improving the performance of single and multi-cluster capabilities of the marl component. Furthermore, the integration between PDLC-DL-MARL and the other PDLC subcomponents, including PDLC-DP for normalized data ingestion, PDLC-CA for context-aware cost modelling, and PDLC-DL-GNNs for resource forecasting has been finalized and validated through comprehensive integration and system testing.

The primary technical advancement of this sub-component in the current deliverable is the realization of multi-cluster capabilities for the Reinforcement Learning agents. By evolving the framework from a localized single-agent model to a decentralized Multi-Agent Reinforcement Learning (MARL) architecture, the system now supports inter-cluster workload optimization. This is facilitated by a secure, auction-based algorithm that allows agents across the federated

network to share action-value (Q-value) estimates to determine the optimal cluster for pod allocation. The focus has been placed specifically on the communication architecture in between the MARL agents.

Multi-cluster sharing of the Q-values generated by PDLC agents has been implemented through the instantiation of a FastAPI instance for each agent within the federated network. This architectural decision is justified by FastAPI's status as a high-performance, modern web framework capable of building APIs with Python 3.8+ that achieve speeds comparable to Node.js and Go. A primary technical benefit for the PDLC-FC-MARL sub-component is FastAPI's native support for asynchronous operations, which utilizes a single-threaded event loop to manage a large volume of concurrent inter-cluster requests without blocking the main execution thread. This non-blocking behaviour is critical for I/O-bound operations, such as the network-intensive task of sharing values across the Edge-Cloud continuum, preventing the agent from becoming unresponsive during high-load traffic spikes. Furthermore, FastAPI leverages Pydantic models for automated data validation and serialization, ensuring that complex data structures like Q-values which represent the action-value function calculating expected returns are guaranteed to meet specified schemas before processing.

The implementation follows a decentralized, non-hierarchical structure where each PDLC FC Agent resides on the control plane of a Managed Cluster (MC) within a defined application neighbourhood. When a new pod arrives and requires allocation, the local agent first writes the generated Q-value to its own local FastAPI instance to record the local "bid" for the workload. To facilitate the auction-based algorithm required for inter-cluster placement, the agent must then share this value with peer agents. This is achieved by utilizing the *grequests* library, which allows the agent to asynchronously multicast the information to the specific IP addresses of other agents in the neighbourhood. By using the *grequests* to issue multiple concurrent HTTP requests, the system can broadcast Q-values to all potential candidates simultaneously, avoiding the latency penalties associated with sequential, synchronous communication. IP addresses are shared through a ConfigMap that is provided by the PDLC-FC-DP subcomponent and they are protected as a Kubernetes Secret, ensuring that no malicious agents can access sensitive information on the neighbouring clusters. This communication protocol supports the privacy-preserving nature of PDLC, as agents exchange only the resultant Q-values rather than sharing sensitive internal model weights or raw infrastructure metadata. Once the multicast is complete and all peer values are received or the pre-configured timeout is reached, the agent with the highest Q-value is selected to allocate the pod, which triggers the writing of *node_recommendations* in the chosen agent cluster.

PDLC-FC-MARL can operate in two distinct working modes, resulting in four possible operational combinations as detailed in Table 8.

Table 8: PDLC-FC-MARL modes

Modules	Single Cluster	Multi Cluster
Inference	Performs inference without the MARL logic (auction-based algorithm).	Performs inference and considers other agents within the same neighbourhood.
Training	Trains a model Online within Single Cluster operations.	Will be addressed in future work.

The sequence diagram of the component remains similar to the one presented in D13, the major revamp has been under the communication logic sequence diagram, which we can see updated in Figure 12.

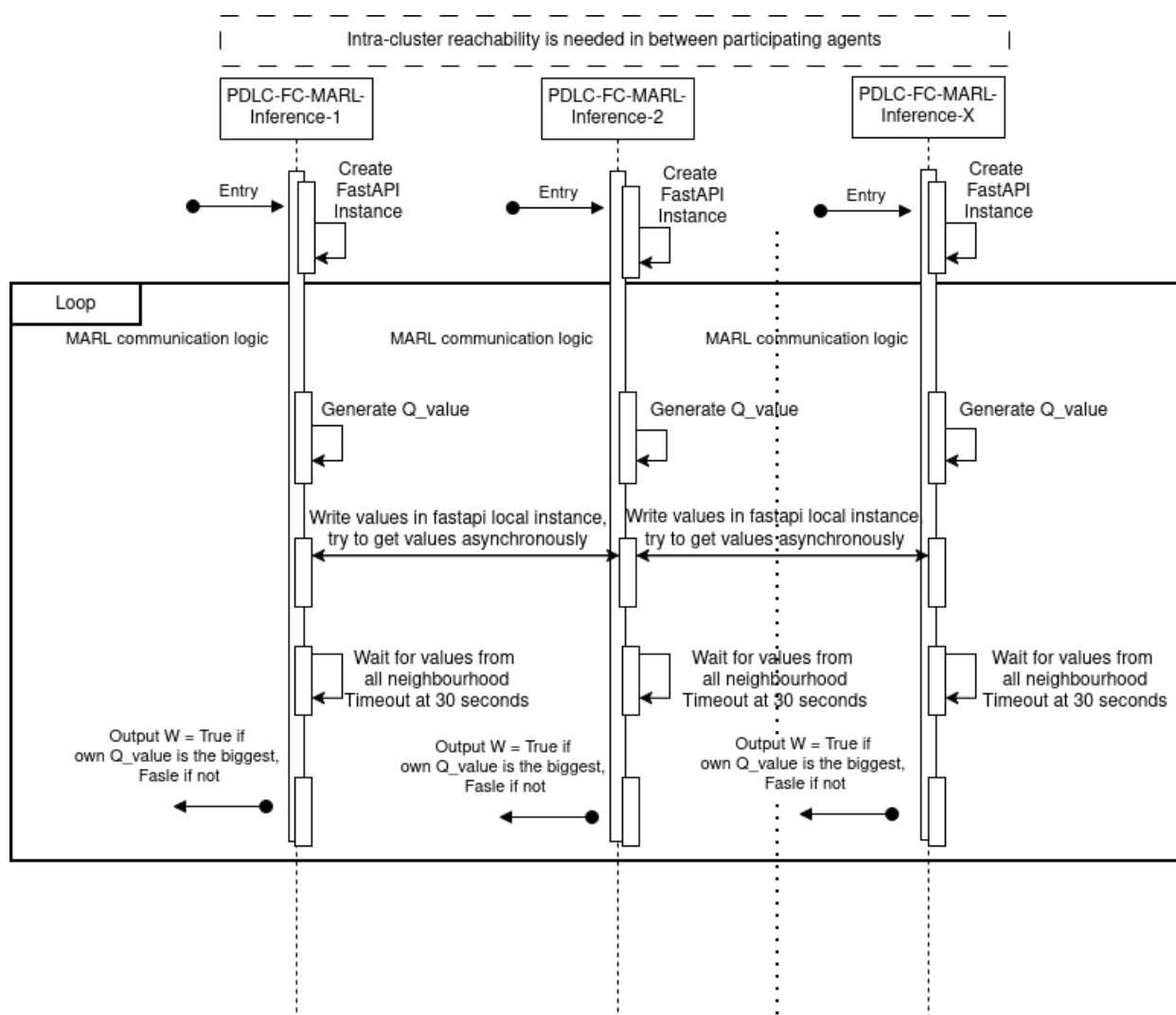


Figure 12: PDLC-MARL-FC components sequence diagram.

3.2.2.4.2 Code structure

Due to the introduction of the FastApi architecture for MARL communication a new module has been added to the code structure in this stage of the project. The file ***fast_api_instance.py*** handles the creation of a local fastapi instance, with memory management for the data written in it, modifications in the rest of the code has been conducted, adding connectivity to the fastapi with web based requests.

Table 9: PDLC-FC-MARL code structure

File	Description
co-deco_online_rl_env.py	Self-containerized gymnasium environment, prepared for Online Training and connected to the CODECO APIs (via InfluxDB and connection to SWM CR).

inference_controller_FC.py	Code containing the inference logic.
influx_utils.py	Code containing functions to connect and extract data from the InfluxDB.
rl_controller.py	Code containing the controller for the MARL component, ensures safety and connection checks are met before starting the component. After that the control flow of the component is engaged, allowing for changes in mode (from training to inference or single to multicluster operations) without restarting the component.
fast_api.py	Implements the local FastAPI interface for each MARL agent, providing endpoints for the asynchronous submission and retrieval of Q-value bids during the inter-cluster auction process.

3.2.2.4.3 Installation Guide

Apart of installing the component with the whole CODECO, two distinct ways are provided to deploy the component locally. First one is the installation integrated with PDLC, which is the recommended installation, in order to do this the user must use the [pdlc_deployment](#) repository. If new docker images want to be created after altering the source code, two different images can be created in this version of the code:

- **MARL image:** docker image which handles the logic of the PDLC-FC-MARL subcomponent.
- **FastAPI image:** docker image that handles the fastapi service for MARL communication, must be ready before deploying the MARL image.

The second one is an individual installation executing the code directly in python without containerizing it, the following steps are recommended:

- **Virtual Environment Setup (Optional but Recommended):** It is advisable to install the component within a virtual environment to ensure dependency isolation and maintain reproducibility. Users may choose either venv or conda for this purpose. For example, using

```
venv: python3 -m venv .venv
```

- **Dependency Installation:** Once the virtual environment is activated, all required Python packages can be installed using the provided requirements.txt file:

```
pip install -r requirements.txt
```

This installation procedure ensures that the component is correctly configured with all necessary dependencies, providing a stable environment for development or deployment.

The entry point of the component is `rl_controller.py`, to switch in between CODECO connectivity and synthetic data connectivity (to test the full pipeline) an additional parameter must be passed when launching the controller, a value of 0 indicates that the component will operate with a direct connection to its CODECO interfaces. Conversely, a value of 1 signifies that the component will use synthetic data, a mode intended for component testing while integration is being finalized:

This will execute MARL connected with CODECO:

```
python rl_controller.py 0
```

This will execute MARL connected with synthetic data:

```
python rl_controller.py 1
```

For additional details regarding setup, configuration, and usage, please refer to the accompanying [README.md](#) in the respective PDLC-FC-MARL repository file provided with the source code.

3.2.2.4.4 *Selected Technologies*

PDLC-FC-MARL is developed using Python 3.10.12 and leverages a variety of libraries and tools, depending on the execution mode. A comprehensive list of all required dependencies is available in the accompanying requirements.txt file included in the codebase:

- **Inference:** During the inference phase, the primary library employed is StableBaselines3-Contrib. Connectivity with other CODECO components is managed through the influxdb-client and Kubernetes libraries, facilitating robust data exchange and system orchestration.
- **MARL communications:** for the Multi-Agent Reinforcement Learning (MARL) communications, the cryptography library is designated for secure message encryption and safeguarding shared values, and the fastapi library has been chosen to simplify communication in between agents in a standardised way. The FastAPI python library is used to handle communication together with the *grequests* library to contact the api services of other agent's clusters.
- **Training:** The training mode leverages a Gymnasium environment for model optimization. It is connected to the CODECO APIs to ensure training in real time.

This framework is intricately connected to CODECO's APIs via InfluxDB for data acquisition and CRD writing for result persistence, thereby enabling real-time data-driven model training. The library suite for training mirrors that of inference, as the training paradigm encapsulates inference functionalities alongside dedicated model training logic.

The CODECO Online Environment is created inheriting from the Gymnasium base Environment, this environment has been created as a point of improvement from the one presented. Pytorch is additionally used for model and data management.

3.2.2.4.5 *Pre-requisites*

Pre-requisites of the framework vary depending on execution mode; the user can specify if the component is deployed by using the InfluxDB as the data input or synthetic data to test the component. The synthetic data mode has no prerequisites, as its purpose is to facilitate isolated testing of the MARL pipeline. Influx mode requires a Kubernetes cluster running CODECO (specifically the PDLC component). To enable decentralized capabilities, you must have multiple clusters running PDLC with full IP connectivity between them.

3.2.2.4.6 *Inputs and Outputs*

All inputs used in our model are standardised via the usage of InfluxDB. In total, we use 6 measurements:

- **Test_integration_metric:** measurement with the necessary data for incoming pods that the component is going to provide *node_recommendations* for, containing the name of the incoming pod, along with the namespace and the requested cpu and ram by the application.



- **Test_node_metric:** measurement with the cpu and ram available per node in the cluster, used to get the necessary information for the state of the MARL component.
- **Greenness_resilience_custom_metric:** measurement with the values provided by PDLC-CA which will be used both as input and as a part of the reward of the models used.
- **ForecastsX:** measurements with forecasts for 5, 15 and 30 minutes. Output provided by GNNs and used as the input of the models. It replaces the incoming node data from DP (test_node_metric) if available, by default the 5 minute forecasts are used.

The other two output of the component remain similar:

- **Comprehensive debugging messages** are provided to help in debugging potential issues that may arise during the last stage of the project.
- **Allocation proposals in the CR file provided as input.** This is done through an attribute added to the SWM CR that indicates how likely a pod should be allocated to a node. For that purpose, a vector of size number of nodes for each pod that needs allocation is used, and for each node the degree of certainty of the pod being allocated to it is *given.fileprovided* as input. This is done through an attribute added to the SWM CR that indicates how likely a pod should be allocated to a node. For that purpose, a vector of size number of nodes for each pod that needs allocation is used, and for each node the degree of certainty of the pod being allocated to it is given.

3.3 NetMA-FC: Network Management and Adaptation

3.3.1 Component Description

NetMA is a network management and adaptation solution that automates the setup of inter-connections for flexible Edge-Cloud operations, while addressing the integration of internet-working control and supporting the needs of diverse network environments (fixed, wireless, cellular) that are expected to be managed by CODECO. This CODECO component handles aspects such as network softwarization, secure data exchange, network performance monitoring and integrated network capability exposure through standard-based mechanisms and Kubernetes APIs.

NetMA and its sub-components, along with the main interfaces towards other CODECO components, are represented in Figure 13. It currently considers the following sub-components:

- **Secure Connectivity (L2S-M).** Secure Connectivity will extend its capabilities to support secure inter-cluster communication across CODECO workloads. The component is currently in its design phase, so the implementation details are defined but not yet disclosed.
- **Network State Monitoring (NSM).** This sub-component allows the use of different network probing mechanisms to bring network status to CODECO. Network status is captured at an underlay and overlay perspective, also from an individual link and path perspective. Updates in this context relate with improvements to the existing single-cluster operation.
- **Network exposure system (Nemesys-FC).** Nemesys brings the networking metrics (underlay and overlay) captured by NSM and exposes them in the CODECO system and will also be responsible for the inter-cluster exposure aspects.

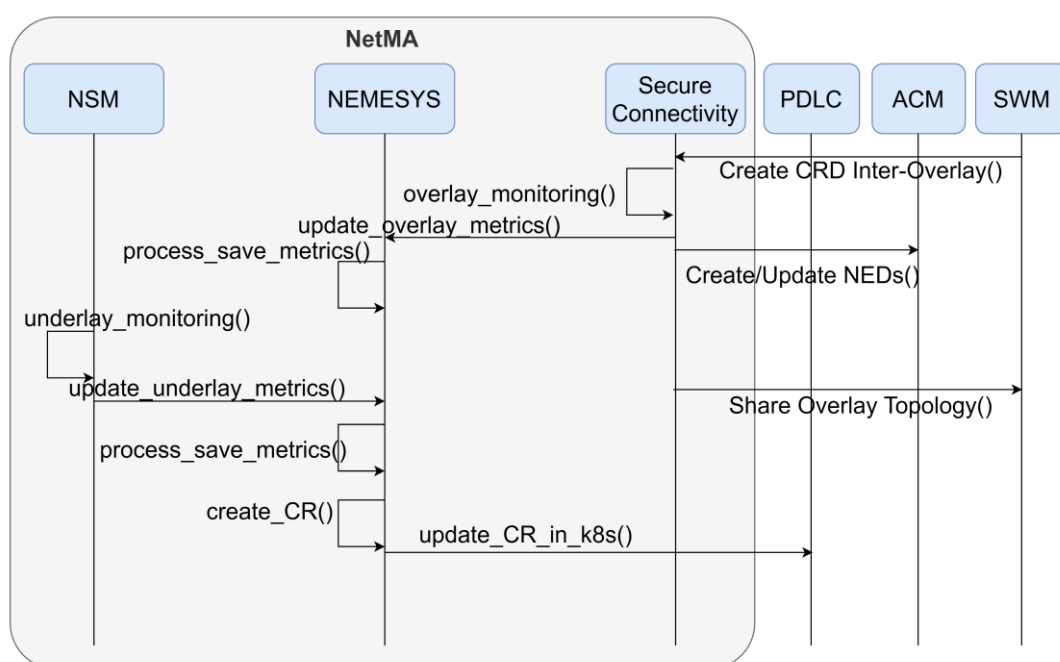


Figure 13: NetMA internal workflows overview.

3.3.2 Sub-components

3.3.2.1 Secure Connectivity-FC

The [NetMA-FC Secure Connectivity](#) subcomponent supports secure inter-cluster communication for CODECO workloads. It serves as the primary connectivity mechanism provided by NetMA. It supports both pre-created and on-demand overlay topologies and integrates with federated scheduling decisions to ensure that network paths align with placement outcomes. Continuous monitoring of inter-cluster links enables runtime adaptation under changing conditions. The overall operational workflow is illustrated in Figure 14.

Secure Connectivity FC operates by combining declarative Kubernetes resources with SDN-based network control, allowing overlay topologies and connectivity paths to be dynamically created, monitored, and adapted. The lifecycle of secure multi-cluster connectivity begins when SWM schedules a workload placement specifying the endpoints in the inter-cluster topology. This is expressed through the creation of a *SliceOverlay Custom Resource*, which describes the clusters participating in the overlay and the logical links that interconnect them.

Once the *SliceOverlay* resource is submitted to the hub cluster, NetMA detects the new configuration and begins its reconciliation process. During reconciliation, NetMA interprets the topology specification and generates the corresponding Network Edge Device (NED) resources. These resources describe how each cluster should connect to the overlay network, including the configuration of the SDN provider and the neighbouring clusters that each node must reach. Similar to the single cluster scenario, the inter-cluster overlay network effectively defines the substrate on which cross-cluster communications will occur.

After the NED resources are created, they are distributed to the appropriate managed clusters using ACM. ACM distributes the NED definitions to the managed clusters according to their role in the overlay topology. When the resources arrive in the managed clusters, the NED components are deployed locally. Each NED initializes the required networking components, including the virtual switch and the inter-cluster VXLAN tunnels necessary to establish connectivity with neighbouring clusters.

Once deployed, the clusters begin reporting information about the realized topology back to NetMA. This feedback includes operational status information, such as whether the NED instances were successfully deployed, as well as runtime metrics about the health and performance of inter-cluster overlay links. These metrics are collected through the monitoring subsystem and provide visibility into network characteristics (e.g., latency and available bandwidth).

With the overlay infrastructure in place, SWM can request connectivity between specific workloads. This is done by submitting a *vlink* configuration, represented by a *SliceNetwork Custom Resource*. The *vlink* specifies the source and destination clusters that must be connected, along with optional performance constraints or requirements such as bandwidth or latency. When receiving a *vlink* request, NetMA carries out a reconciliation procedure. It analyses the existing overlay topology and computes a suitable end-to-end path between the specified clusters. Once a valid path is determined, NetMA interacts with the Inter-Domain Connectivity Orchestrator (IDCO) to translate the logical connectivity request into concrete network configurations.

The IDCO controller then programs the data plane by installing the necessary OpenFlow rules across the Network Edge Devices located at the boundaries of each cluster. These rules define how traffic should be steered across the overlay network, effectively stitching together the intra-cluster and inter-cluster segments required to create a continuous communication path.

After the configuration is applied, the workloads deployed in the different clusters can communicate through the overlay network using the provisioned *vlink*. The resulting path provides secure and isolated connectivity across the federated cluster environment, while remaining continuously monitored so that the system can adapt to changing network conditions if necessary.

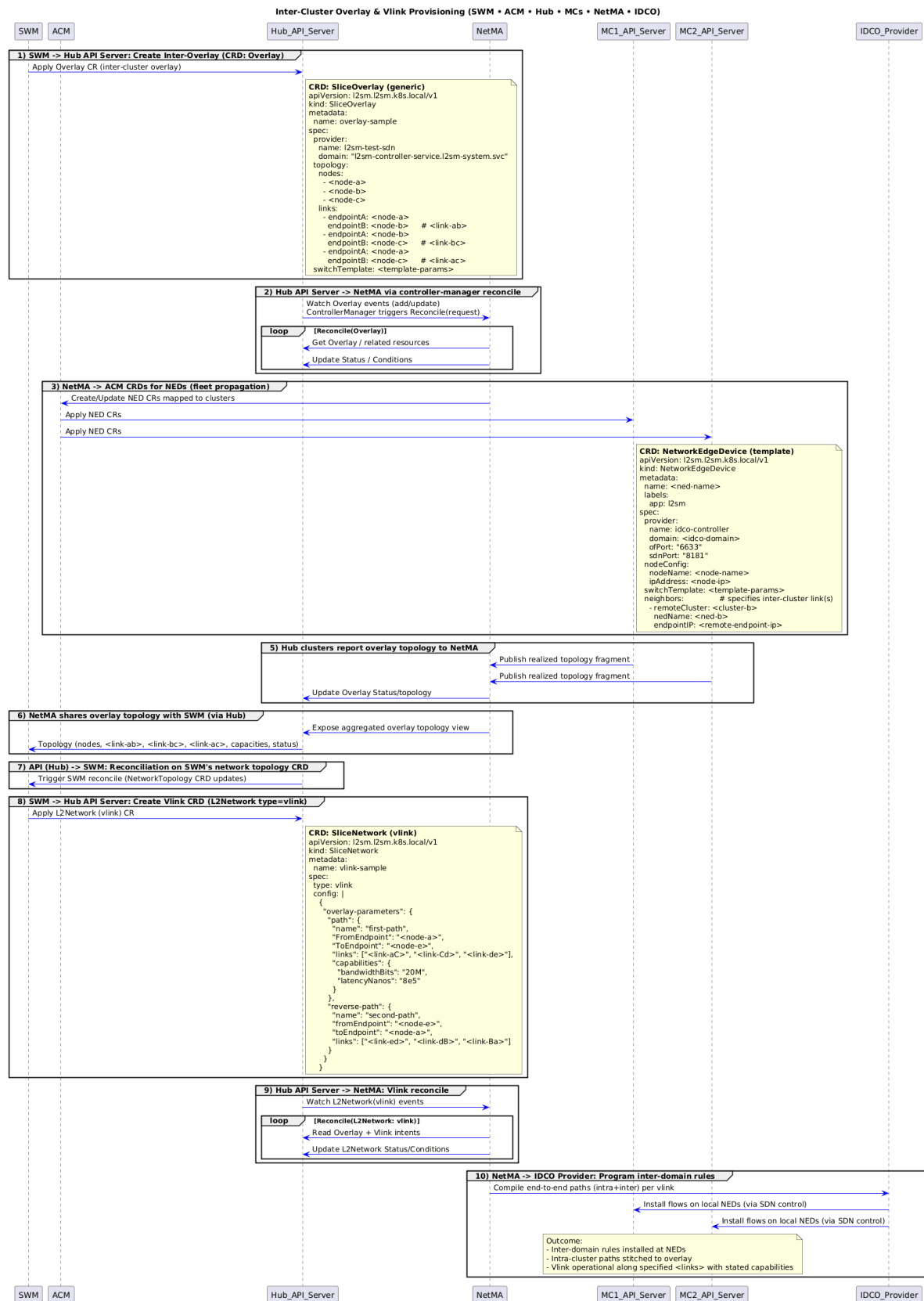


Figure 14: NetMA-FC Secure Connectivity Operational Workflow.

3.3.2.1.1 Code Structure

The Secure Connectivity FC sub-component main source code files are briefly summarized in Table 10.

Table 10: Secure-Connectivity FC code structure.

File	Function
deployments/codeco-deployment.yaml	Kubernetes YAML file for deploying the full component in the hub cluster, including all necessary resources and CRD specifications.
Dockerfile	Dockerfile used for building the kubernetes manager docker image from scratch for the IDCO
Config/codeco/*.yaml	Kustomize resources for creating a customized Kubernetes installation file for the CODECO deployment
api/v1/*.go	Defines the Custom Resource Definitions (CRDs) for the Secure Connectivity FC component and their associated Kubernetes API scaffolding, including the schema for configuration inputs (Spec) and operational status reporting (Status), enabling Kubernetes-native resource management and reconciliation
Internal/controller/*.go	Code implementing the reconciliation logic of the Secure Connectivity FC controllers, watching the CRDs defined in api/v1 and driving the system toward the desired state by translating high-level overlay topology and connectivity intents into concrete resource operations and status updates.
Makefile	Automation file for managing the subcomponent development process through make instructions
IDCO/pom.xml	POM file for building the IDCO Provider SDN Distributed Controller, with maven.
IDCO/Dockerfile	Main dockerfile for building the IDCO Provider SDN Controller docker image
IDCO/src/main/java/org/l2sm/vlinks/rest/NetworkManagement.java	REST API code for IDCO virtual network management, providing endpoints for health monitoring, network lifecycle operations (create, retrieve, delete), and port membership management within virtual networks
IDCO/src/main/java/org/l2sm/vlinks/app/IDCOVLinkManager.java	Implements the core SDN ONOS application logic for IDCO, managing virtual L2 networks through distributed state, intent-based connectivity, and dynamic host

File	Function
	learning. Additionally, it handles network lifecycle and endpoint membership management

3.3.2.1.2 Selected technologies

The NetMA Secure Connectivity FC subcomponent is implemented as a set of three loosely coupled microservices, each addressing a specific functional responsibility. This modular design improves scalability, maintainability, and alignment with cloud-native deployment practices.

3.3.2.1.2.1 IDCO Provider SDN Controller

The [SDN Controller](#) (IDCO) is implemented as a Java 17 application based on ONOS. It is responsible for inter-cluster network control and data-plane coordination. To support fault tolerance and controller distribution, the SDN Controller relies on:

- Atomix 3.0: a framework for building fault-tolerant distributed systems based on the RAFT consensus protocol. Atomix enables consistent state sharing across multiple SDN Controller instances, allowing the control plane to be distributed while maintaining synchronized, centralized control logic.

This technology choice supports high availability and consistency across federated deployments.

3.3.2.1.2.2 IDCO Controller Manager

The IDCO Controller Manager acts as the integration layer between NetMA subcomponents (including Nemesys FC), and external CODECO components such as SWM and ACM. It exposes and manages control-plane interactions and lifecycle coordination.

This microservice is implemented in Go 1.25 as a Kubernetes Controller Manager, leveraging native Kubernetes extensibility mechanisms. It is based on the following technologies:

- Kubebuilder (v4.11): a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs), enabling declarative configuration and reconciliation of inter-domain connectivity resources.
- OCM API: used to interact with managed clusters via the ACM FC, supporting federated and multi-cluster operation.

The use of Kubernetes-native patterns ensures seamless integration with existing orchestration and management workflows.

3.3.2.1.2.3 i-LPM Exporter

The inter-cluster LPM (i-LPM) Exporter is implemented in Go 1.25 and is responsible for exporting inter-cluster network metrics.

It is based on Prometheus, enabling standardized metric collection and exposure. The exporter supports continuous monitoring of inter-cluster links and provides the data required for runtime adaptation under changing conditions.

3.3.2.1.3 Pre-requisites

The NetMA Secure Connectivity FC subcomponent requires a running and operational Kubernetes cluster with a Kubernetes version 1.25 or later. The cluster must allow deployment of

controller-based microservices and support standard Kubernetes networking and Custom Resource Definitions.

For inter-cluster operation, each node in the hosting cluster must expose a set of open network ports to enable communication with managed clusters and external network components. By default, the availability of the following ports are required:

- 30663: used for OpenFlow 1.3 communication with the NEDs.
- 30808: used for HTTP-based communication with managed clusters. When a workload pod is scheduled in a managed cluster, the corresponding controller manager sends a confirmation message to this port.

The default port configuration can be modified in the following file:

```
./config/codeco/idco-service-patch.yaml
```

Although not strictly required, the Secure Connectivity FC is designed to operate in a multi-cluster environment. For correct baseline operation, it is expected that multiple managed clusters are available and that each managed cluster has single cluster Secure Connectivity deployed and operational.

3.3.2.1.4 Installation guide

To deploy the NetMA Secure Connectivity FC subcomponent, the following command must be executed from the root directory of the subcomponent:

```
kubectl apply -f ./deployments/codeco-deployment.yaml
```

This command deploys all required Kubernetes resources, including deployments, services, configuration objects, and Custom Resource Definitions.

Upon successful execution, the Secure Connectivity FC components are instantiated in the he-codeco-netma Kubernetes namespace and start operating automatically without additional manual configuration.

3.3.2.1.5 Inputs and Outputs

The Secure Connectivity FC consumes configuration inputs expressed as Kubernetes Custom Resources. These resources define the desired inter-cluster connectivity and monitoring behaviour.

- **SliceOverlay:** Defines the overlay topology data plane spanning multiple managed clusters. The resource can be updated dynamically, allowing the overlay topology to be reconfigured at runtime. The key configuration field is `.spec.topology`, which captures both cluster membership and inter-cluster link definitions. Monitoring outputs are configured to target both the SWM FC Network Topology resource and the Nemesys FC Overlay Status. This resource specifies:
 - The set of managed clusters participating in the overlay, identified by cluster IDs.
 - The logical links between clusters, describing how Network Edge Devices (NEDs) should interconnect.
 - Monitoring configuration for i-LPM, including how inter-cluster links are observed and where monitoring results are published.

- **SliceNetwork:** Represents a logical network spanning multiple managed clusters, used by distributed workloads to communicate. This resource specifies the pair of target clusters where the network must be instantiated and made available to the workloads.

Based on the provided inputs, the Secure Connectivity FC produces the following outputs:

- **Inter-cluster network realization:** The component deploys and configures the required networking resources in the managed clusters to enforce the desired connectivity state. This process is executed through the ACM FC component, ensuring consistent deployment across clusters.
- **Network topology feedback for scheduling:** The component publishes a SWM FC Network Topology Custom Resource populated with i-LPM metrics collected from the overlay. This information is used to enhance federated scheduling decisions.
- **Export of metrics for NetMA coordination:** The component exposes a dedicated endpoint that allows Nemesys FC to retrieve overlay-level metrics and combine them with other NetMA FC measurements for higher-level analysis and optimization.

3.3.2.1.6 Expanding the Codebase and New Features

Expanding Secure Connectivity FC typically involves modifications in three main areas. First, developers may extend the CRD definitions in `api/v1/*.go` to introduce new configuration parameters, such as additional topology attributes, monitoring settings, or connectivity constraints. This requires updating the Spec or Status fields and regenerating the Kubernetes manifests and client code so that the new schema is recognized by the system.

Second, the controller logic in `internal/controller/*.go` must be updated to process these new fields. The reconciliation functions interpret the desired state expressed in the CRDs and translate it into actions such as creating or updating network resources, invoking IDCO for connectivity configuration, or updating status fields to reflect the observed system state.

Finally, developers may need to adjust integration and deployment components, including interactions with IDCO, NED configuration behaviour, monitoring exports, or Kubernetes deployment files (e.g., in `config/` or `deployments/`). These updates ensure that the new functionality is correctly integrated with the rest of the NetMA and CODECO infrastructure.

3.3.2.2 NSM

3.3.2.2.1 NetMA-nsm-mon

Within the NetMA sub-component [NSM](#), `netma-nsm-mon` is responsible for bringing underlay network state into CODECO. It continuously measures and exposes link-level and node-level networking metrics to the orchestration layer.

The current implementation consists of two complementary monitoring slices:

1. **Active monitoring**, [netma-nsm-mon](#) (legacy probing using `Netperf/iPerf/ping/tcpdump`)
2. **Passive monitoring**, [netma-nsm-mon-passive](#) (eBPF-based real-time observation using `ePPing`)

The passive slice represents the latest evolution of `netma-nsm-mon` and is the recommended operational mode for federated cluster environments. The active probing is described in D13.



Table 11: Key differences on the update from active to passive probing.

Aspect	Active Mode	Passive ePPing Mode
Traffic Injection	Yes	No
RTT Source	netperf TCP_RR	Kernel-level RTT (eBPF)
Bandwidth Source	iPerf	Observed real traffic
Overhead	Medium/High	Low
Real-Time	Periodic	Continuous
Production Suitability	Benchmarking	Recommended

All collected metrics are exposed to upstream CODECO components via Kubernetes ConfigMaps and are consumed by PDLC (learning) and SWM (scheduling).

Netma-nsm-mon has been based on the existing open-source code of the [K8s netperf](#) licensed under Apache 2.0. and focuses on bringing active network probing to CODECO. The metrics considered are provided in Deliverable D31 and are again presented in Annex I.

The current implementation provides CODECO with the following networking metrics: i) bandwidth; ii) latency; iii) packet loss; iv) jitter; v) node degree; vi) link failures; vii) link energy; viii) flow energy, and access to other interface statistics.

- **Periodic Probing:** Probes are configured to run at set intervals, evaluating link/channel performance (underlay).
- **Metric Collection & Aggregation:** Raw probe results such as RTT, jitter, energy consumption are aggregated and averaged.
- **Exposing Custom Resources (CRs):** Collected metrics are made available via Kubernetes CRs, enabling consumption by other NetMA components such as PDLC (learning) and SWM (scheduling).
- **Integration with Nemesys:** collected metrics are integrated with the NetMA Nemesys component.

Netma-nsm-mon-passive continuously samples network behaviour inside Kubernetes clusters using:

- eBPF-based ePPing RTT probes
- Kernel-level counters
- nftables-based traffic aggregation
- Flow-level inspection

It exposes the collected metrics via ConfigMaps. The following metrics are currently exposed:

- Bandwidth
- Latency (RTT)
- Packet loss
- Jitter
- Network interface statistics
- Network energy estimation (link and IP flow level).

Unlike the active probing mode, passive monitoring does not inject synthetic traffic. Instead, it observes real flows and derives metrics with minimal overhead. The monitoring stack consists of:

- **ePPing DaemonSet** (privileged, BPF-enabled)
- **NSM Controller** (gRPC aggregation service)
- **ConfigMap exporter**

Each node runs an ePPing agent. Metrics are aggregated centrally and written as cluster-level and node-level ConfigMaps.

High-level workflow:

1. ePPing agent attaches to kernel via eBPF.
2. Passive RTT and flow-level metrics are collected.
3. Agent reports metrics via gRPC to the NSM controller.
4. Controller aggregates metrics per node and cluster.
5. Results are written to Kubernetes ConfigMaps.
6. PDLC and SWM consume metrics via Kubernetes API.

No direct RPC coupling with other CODECO components exists. ConfigMaps serve as the integration boundary.

3.3.2.2.1.1 Sequence Diagram and Operation Workflow

The sequence diagram is provided in Figure 15: The deployment of netma-nsm-mon is represented in Figure 16.

- ePPing agent attaches to kernel (BPF)
- Passive metrics are extracted per node
- Agent sends metrics to controller via gRPC.
- Controller aggregates metrics.
- Controller writes:
 - cluster-metrics ConfigMap
 - flow-event-* ConfigMaps
 - netperf-metrics-* ConfigMaps
- CODECO components (PDL, SWM, Nemesys) consume metrics via Kubernetes API

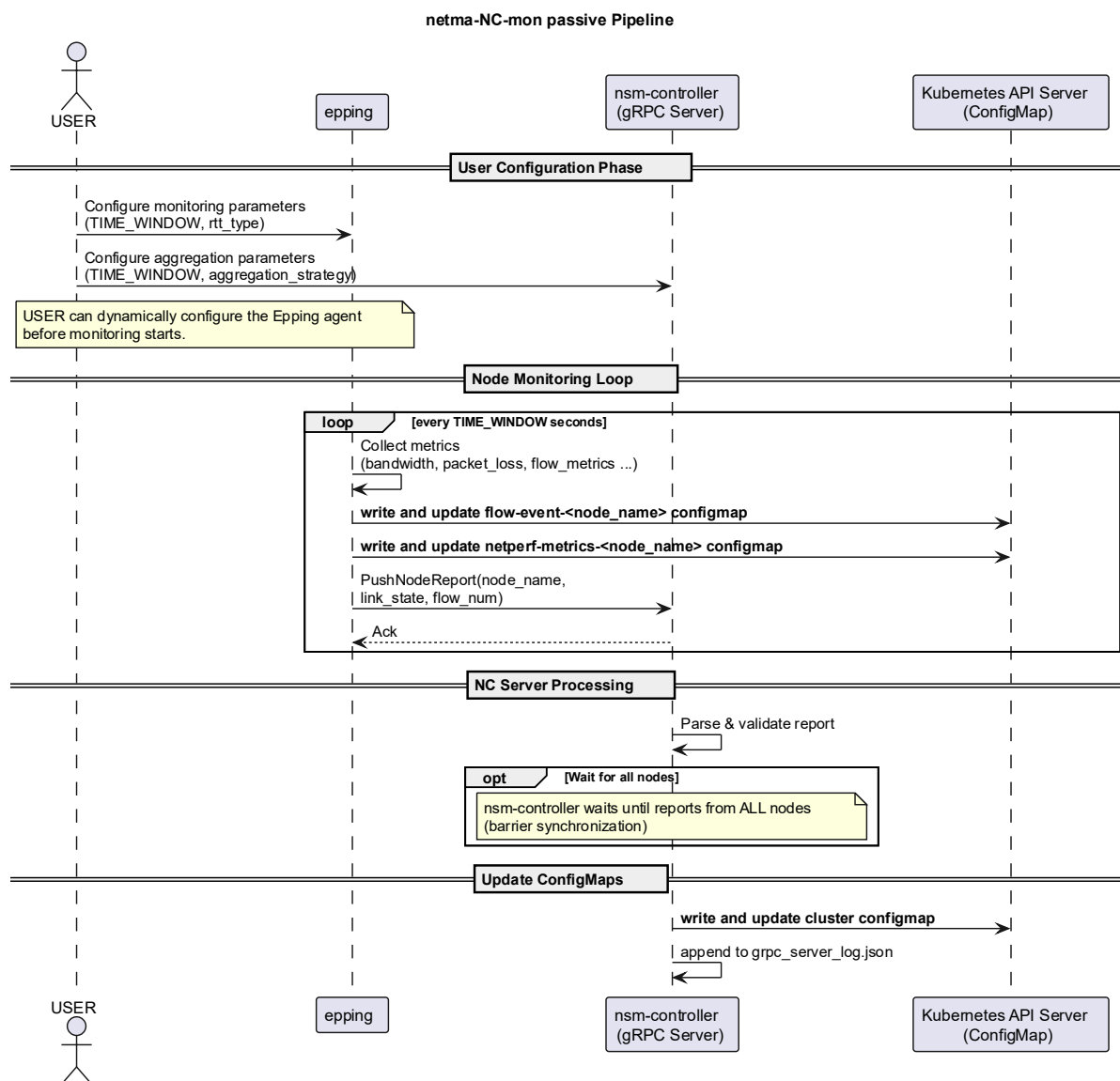


Figure 15: netma-nsm-mon sequence diagram.

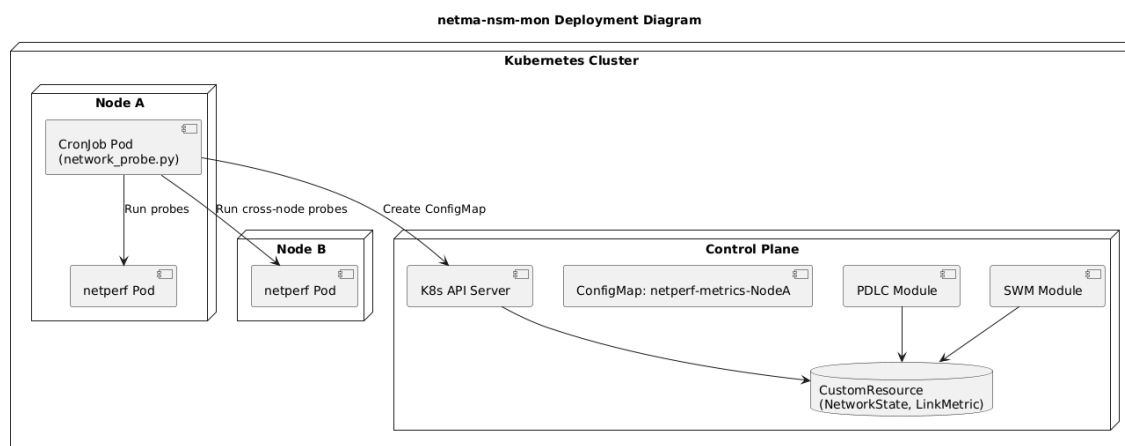


Figure 16: netma-nsm-mon pod placement across control and service planes.

3.3.2.2.1.2 Implementation Details

This section describes how the passive monitoring slice of netma-nsm-mon is implemented at runtime, including kernel integration, agent-controller communication, Kubernetes integration, and metric exposure to CODECO components.

The passive monitoring design follows three core principles:

1. Kernel-level observability with minimal overhead
2. Strict separation between collection and aggregation
3. Kubernetes-native integration via ConfigMaps

Each Kubernetes node runs a privileged DaemonSet pod containing the ePPing-based monitoring agent.

Kernel Integration

The agent attaches to the Linux kernel using eBPF (CO-RE enabled). The implementation requires:

- Linux kernel ≥ 5.2
- BTF (BPF Type Format) support
- Privileged container access

The eBPF program:

- Hooks into networking stack events
- Extracts round-trip latency (RTT)
- Observes flow characteristics
- Samples traffic metadata

Unlike active probing, no synthetic traffic is injected. Metrics are derived from real application traffic.

The main execution logic resides in: [epping/passivemonitor.py](#).

Source Node (worker) level passive monitoring (epping agent)

At runtime, the agent:

1. Initializes Kubernetes discovery (node name, namespace, labels).
2. Attaches eBPF probes.
3. Periodically samples:
 - RTT distributions
 - Packet counters
 - Bandwidth usage
 - Loss estimates
 - Interface statistics
4. Computes derived metrics such as:
 - Jitter (RTT variance)



- Energy estimates (based on byte counts and modelled constants)
- 5. Packages metrics into structured messages.
- 6. Sends metrics via gRPC to the NSM controller.

Asynchronous execution is handled using Python's `asyncio` to ensure non-blocking metric collection and reporting.

gPRC Communication Schema

Communication between node agents and the central controller is implemented via gRPC.

The message schema is defined in [controller/nsm-mon.proto](#)

This schema defines:

- Node-level metric structures
- Flow event structures
- Aggregated summaries

The agent serializes collected metrics into typed messages and sends them to the controller endpoint exposed via a Kubernetes Service. This design ensures:

- Versioned metric schema
- Structured, extensible payloads
- Clear separation between collection and aggregation

Centralized aggregation

The aggregation service runs as a Deployment: [controller/server.py](#)

Its responsibilities are:

- Accept metric streams from all nodes
- Maintain in-memory state for current cluster view
- Aggregate per-node metrics into cluster-level summaries
- Generate Kubernetes ConfigMaps

The controller writes:

- cluster-metrics
- flow-event-<node>
- netperf-metrics-<node>

The Kubernetes Python client is used to create or update ConfigMaps atomically.

The controller does not perform scheduling logic. It strictly exposes network state.



Kubernetes integration

The monitoring stack uses native Kubernetes constructs:

- **DaemonSet** — one monitoring agent per node
- **Deployment** — centralized controller
- **Service** — gRPC endpoint
- **ConfigMaps** — metric export interface
- **RBAC** — minimal required permissions

No CRDs are defined within this component. Instead, ConfigMaps serve as the integration boundary with higher-level CODECO components. This ensures loose coupling and simplifies failure recovery.

Federation Behavior

In a federated CODECO environment:

- Each Managed Cluster runs its own netma-nsm-mon-passive stack.
- Metrics remain cluster-scoped.
- Hub-level orchestration components (ACM, PDLC, SWM) consume metrics via OCM propagation mechanisms.

There is no cross-cluster metric sharing at the monitoring layer. Federation logic is handled upstream.

Failure handling and resilience

The implementation handles:

- Agent restarts (DaemonSet reconciliation)
- Controller restarts (stateless aggregation rebuild)
- Temporary node disconnections
- Partial metric availability

Because ConfigMaps are continuously updated, upstream components always consume the latest consistent state. No historical time-series storage is maintained within netma-nsm-mon-passive. Long-term metrics are delegated to Prometheus or external monitoring systems.

3.3.2.2.1.3 Implementation Details

The ePPing agent requires:

- Privileged container access
- Host networking
- eBPF attachment permissions



RBAC is scoped to:

- ConfigMap write access
- Node discovery

No cluster-admin privileges are required beyond what is necessary for BPF and ConfigMap updates.

3.3.2.2.1.4 Extension Suggestions

Developers can extend the implementation by:

- Adding new metrics to the .proto schema
- Enhancing energy modelling logic
- Introducing adaptive sampling intervals
- Exporting metrics as CRDs instead of ConfigMaps
- Integrating with Prometheus exporters

All extensions should preserve the separation between:

- Collection (agent)
- Aggregation (controller)
- Consumption (PDLC / SWM)

3.3.2.2.1.5 Metrics' Models

The current implementation of netma-nsm-mon relies on passive probing using ePPing (eBPF-based RTT extraction) rather than synthetic traffic generation. Metrics are derived from real traffic observed at the kernel level and continuously sampled on each node.

Each Kubernetes node runs an ePPing-based monitoring agent (DaemonSet), which attaches to the networking stack via eBPF and extracts underlay performance indicators. Metrics are aggregated at cluster level and exported via Kubernetes ConfigMaps.

Unlike the previous active probing model, no artificial traffic is injected. All metrics reflect actual runtime behaviour.

Latency measurement model

Latency is derived passively using ePPing, which measures RTT by observing TCP flows in the kernel networking stack. The model reflects real application traffic latency rather than synthetic TCP_RR benchmarks. The agent captures:

- Round-trip time (RTT)
- Percentile distributions (P50, P90, P99)
- Flow-level latency characteristics

Example exported metric format:

```
latency.p90.milliseconds.origin.nodeA.destination.nodeB=4.12
```

Bandwidth measurement model

Bandwidth is estimated from observed byte counters collected via:

- eBPF hooks
- nftables flow accounting
- Kernel interface statistics
- The agent samples traffic rates over fixed intervals and computes average throughput per node pair.

Example format:

```
bandwidth.mbps.origin.nodeA.destination.nodeB=125.4
```

This metric reflects effective traffic rates under actual load conditions.

Packet loss model

Packet loss is estimated using:

- TCP retransmission observations
- Socket-level statistics (ss)
- Flow-level anomalies detected via eBPF

Rather than active ping failure counting, loss estimation is inferred from retransmissions and missing acknowledgments.

Example format:

```
packet.loss.rate.origin.nodeA.destination.nodeB=0.0023
```

Jitter Model

Jitter is computed as the statistical variation of RTT samples observed via ePPing. It is derived from:

- Variance across RTT distributions
- Short-term latency fluctuations

Example format:

```
latency.jitter.milliseconds.origin.nodeA.destination.nodeB=0.84
```

This captures network stability rather than absolute delay.

Node degree

To measure the node degree, function `measure_node_degree(...)` pings the neighbour nodes and counts successful replies as “degree”:

```
node.degree.of.nodeA=3
```

Link failure

Link failure is supported by `measure_link_failures(...)`. It pings all other nodes every 30s for N minutes, recording how many times each node failed to respond.

Link energy

Energy consumption is estimated from observed traffic volumes using modelled energy-per-byte coefficients. The agent:

- Samples byte counters
- Applies calibrated constants
- Computes energy per node pair

Example format:

```
link.energy.consumption.origin.nodeA.destination.nodeB=4021.45.microwatts
```

These tests are run between every pair of nodes using dedicated host-network pods (usually 1 per node). Each result is written as a simple key-value pair as shown in Figure 17, `network_metrics.txt`.

```
netperf.p90.latency.milliseconds.origin.nodeA.destination.nodeB=12.3
iPerf.bw.Gbits.Sec.origin.nodeA.destination.nodeB=5.3
```

Figure 17: example of output for the probes generated by netma-nsm-mon.

The `network_metrics.txt` file are then uploaded as a K8s ConfigMap on each node, via the method `create_or_update_configmap(configmap_name, namespace, file_path)`.

Flow Energy Model

Flow energy represents the estimated energy cost associated with application-level data flows, rather than just link-level traffic between nodes. It provides observation on the energy consumed by the transmission of an IP flow. In the passive implementation, flow energy is derived from:

- eBPF-observed flow statistics (per 5-tuple or aggregated flow)
- Byte counters per source/destination pair
- Flow duration
- Modelled energy-per-byte coefficients, as defined by Feeney et al. [5]

Unlike link energy, which aggregates all traffic between two nodes, flow energy isolates the contribution of specific application flows.



Example:

```
flow.energy.consumption.origin.nodeA.destination.nodeB.app.serviceX=812.34.microwatts
```

Interface statistic model

In addition to flow-level metrics, the agent samples kernel interface statistics, including:

- RX/TX packet counters
- Error counters
- Drop statistics
- Queue lengths

These metrics provide node-level context used by PDLC and SWM.

Execution Model

Each node runs a privileged DaemonSet pod that:

- Attaches eBPF programs to the kernel.
- Observes live traffic.
- Aggregates metrics per interval.
- Sends structured results via gRPC to the NSM controller.

The NSM controller aggregates node-level data and writes:

- cluster-metrics ConfigMap
- flow-event-* ConfigMaps
- netperf-metrics-* ConfigMaps

No flat files are used in the passive implementation.

3.3.2.2.1.6 Code Structure

The main source code files of netma-nsm-mon-passive are summarized in Table 12. The codebase cleanly separates responsibilities into three layers.

Deployment Layer This layer provisions infrastructure and deploys monitoring components.

- [deploy.sh](#)
- [kind-config.yaml](#)
- Kubernetes YAML manifests

Node-Level Monitoring Layer (ePPing Agents). Located under [epping/](#), this layer is horizontally scalable (one instance per node via DaemonSet). This layer is horizontally scalable (one instance per node via DaemonSet):

- Attaches to kernel via eBPF



- Observes real traffic
- Extracts underlay metrics
- Sends structured results to controller

Aggregation & Exposure Layer (Controller). Located under [controller/](#), this layer is logically centralized per cluster, provisions infrastructure and deploys monitoring components.

- Aggregates per-node metrics
- Computes cluster-level summaries
- Writes Kubernetes ConfigMaps
- Exposes data to PDLC and SWM

Table 12: Description of the main netma-nsm-mon source code files.

Path / File	Component Type	Purpose	Runtime Role in Monitoring Pipeline
deploy.sh	Deployment Script	Automates end-to-end setup (KinD cluster creation, namespace creation, DaemonSet and controller deployment, RBAC configuration).	Used for quick-start and testing. Sets up the full passive monitoring stack in a reproducible environment.
kind-config.yaml	KinD Configuration	Defines cluster topology for local development (control-plane + worker nodes).	Provides deterministic multi-node environment for testing inter-node monitoring.
nsm-epping.yaml	Kubernetes Manifest (DaemonSet + RBAC)	Deploys the ePPing-based passive monitoring agent as a privileged DaemonSet.	Ensures one monitoring agent runs per node. Grants necessary BPF and host-network access.
nsm-controller.yaml	Kubernetes Manifest (Deployment + Service + RBAC)	Deploys the centralized NSM controller service and exposes its gRPC endpoint.	Aggregates metrics from agents and writes cluster-level ConfigMaps.
controller/	Aggregation Layer	Contains the central metric aggregation and export logic.	Receives metrics from agents and publishes them to Kubernetes.
controller/server.py	gRPC Server	Implements the aggregation service. Receives node-level metrics and writes ConfigMaps.	Core integration bridge between monitoring layer and CODECO.
controller/nsm-mon.proto	Protocol Definition	Defines gRPC message schema for agent-to-controller communication.	Ensures typed, versioned metric exchange between agents and controller.
controller/Dockerfile	Container Definition	Builds the NSM controller container (Python + gRPC + Kubernetes client).	Produces the runtime image for centralized aggregation.
epping/	Passive Monitoring Agent	Contains the eBPF-based monitoring logic executed per node.	Observes underlay traffic and extracts KPIs.

Path / File	Component Type	Purpose	Runtime Role in Monitoring Pipeline
epping/pas-sivemonitor.py	Agent Runtime	Main loop for metric collection. Uses eBPF (ePPing), nftables, and kernel counters.	Collects RTT, bandwidth, loss, jitter, energy; sends to controller.
epping/utils.py	Utility Library	Helper functions for Kubernetes discovery, metric normalization, ConfigMap handling, nftables integration.	Supports agent runtime with cluster awareness and metric formatting.
epping/Dockerfile	Container Definition	Builds the ePPing agent container, including BPF runtime and Python environment.	Produces node-level DaemonSet image.
diagram/	Documentation Assets	Architecture diagrams and UML representations.	Used for documentation and design validation (no runtime role).

3.3.2.2.1.7 Selected Technologies

The current version of netma-nsm-mon has considered the following technologies:

- **K8s netperf⁴:** K8s Netperf is a benchmarking tool designed to measure various aspects of networking performance, focusing on bulk data transfer and request/response performance using TCP or UDP with the Berkeley Sockets interface. It supports tests over IPv4 and IPv6, and can be used on multiple platforms, including Unix, Linux, and Windows. It provides an effortless way to test network performance between pods using various metrics. To deploy, apply the provided YAML file which sets up daemon sets and pods to measure host-to-host, pod-to-pod (intra- and inter-node), and pod-to-service (ClusterIP) performance. It is currently used for network latency testing based on the TCP protocol.
- **iPerf3:** In the current netma-nsm-mon version it is used for bandwidth and packet loss testing based on the UDP protocol.
- **ICMP Ping:** It is currently used for connectivity testing to monitor node degree and link failure states.
- **TCPdump:** TCPdump is a powerful command-line packet analyzer used for network troubleshooting and security auditing. It captures and displays the contents of network packets transmitted over a network in real-time, allowing users to diagnose network issues and understand the data being transferred. TCPdump supports filtering criteria to capture specific types of traffic, making it a versatile tool for network administrators and security professionals. It can analyze protocols such as TCP, UDP, and ICMP, providing detailed insights into network activity. Used for packet capture and analysis to calculate link energy.
- **Python asyncio.** Python's asyncio for asynchronous operations, ensuring simultaneous and orderly recording of multiple network parameters.

3.3.2.2.1.8 Pre-requisites

An active cluster with at least two worker nodes.

3.3.2.2.1.9 Installation Guide

The overall set up of netma-nsm-mon can be found in its [readme](#).

⁴ <https://github.com/vtrocelab/netperf-2.7.0/tree/master>

3.3.2.2.1.10 Inputs and Outputs

- Input to netma-nsm-mon are the networking underlay metrics being monitored, provided in Annex I.
- Outputs are the generated networking metrics (ConfigMap).

3.3.2.2.2 NetMA-nsm-npp

3.3.2.2.2.1 Code Structure

The main source code files of netma-nsm-npp are summarized in Table 13.

Table 13: Description of the main netma-nsm-npp source code files.

File	Function
Dockerfile	Creates a Docker image for the probe.
daemonset.yaml	Configuration file for deploying probing mechanisms in the different nodes of a cluster. One probe is deployed for each node. If additional nodes are created on-demand, new probes will be created.
data_communication.py	Source file to handle different configuration options to expose the metrics gathered for the probe.
network_probe.py	Main source file of the probe, which contains the logic of the measurements, and the execution of the communication and exposure of the metrics.

3.3.2.2.2.2 Pre-requisites

There are three types of dependencies to be handled to deploy this component:

- **Hardware dependencies:** There are no specific hardware requirements beyond those already defined for the installation of the CODECO Framework. The component has minimal computational footprint and can run on standard control-plane or worker nodes.
- **Software dependencies:** this code requires to have installed Kubernetes, Docker and Python 3.x installed in the machines to be deployed. Additionally, a server (such as Apache or Nginx) must be installed and configured in each node, as the system exposes an HTTP endpoint on port 80. This setup ensures that the NSM's exposed endpoints are reachable and compliant with standard HTTP communication patterns expected by the framework.
- **Libraries dependencies:** There are some needed libraries to be installed to make this code run (python [requirements.txt](#)).

3.3.2.3 Network Exposure: Nemesys-FC

Nemesys-FC serves as the North-Bound Interface (NBI) within the NetMA component, responsible for gathering and formatting topology and performance data from both overlay and underlay network probes. This processed information is then exposed to other CODECO components, such as the PDLC and ACM, enabling seamless integration and communication across the system. In multi-cluster environments, Nemesys instances must federate by exchanging topological data with their counterparts in each cluster, forming a Peer-to-Peer (P2P) mesh that ensures traceability and a unified network view. Nemesys-FC is represented in Figure 18.

To facilitate this exchange, an ALTO server is implemented, collecting inter-cluster performance metrics and generating cost maps using Dijkstra's algorithm. Each Nemesys periodically shares and receives metrics like latency, bandwidth, packet loss, and energy from its peers, allowing the creation of a comprehensive and consistent network map. The main metric, typically latency, guides optimization decisions, while multi-cost maps allow for a realistic representation of network paths by capturing multiple metrics simultaneously. This federated information is formatted according to the cluster's CRD and made available to the PDLC for topology and performance management.

3.3.2.3.1 Sequence Diagram and Operation Workflow

To support these functionalities, a modular architecture has been adopted for the sub-component. This design includes two dedicated modules responsible for reading information from the overlay and underlay network probes, respectively. Although the overlay and underlay networks coincide—since they reference the same node pairs—the design enables integration of both views into a unified multi-metric graph. As illustrated in Figure 18, this graph structure incorporates multiple performance metrics, with overlay latency selected as the primary metric.

The main module of Nemesys, the ALTO core, will retain this view and create a second one to which it will add the information received from the federation API. It is also responsible for creating the maps that will be exchanged and subsequently formatting the CR.

The ALTO Core also communicates with the federation API to send it the local network map after each update. This map is the information exchanged between the federation members, forwarding just the local views and its updates. Figure 19 provides the workflow sequence diagram for Nemesys.

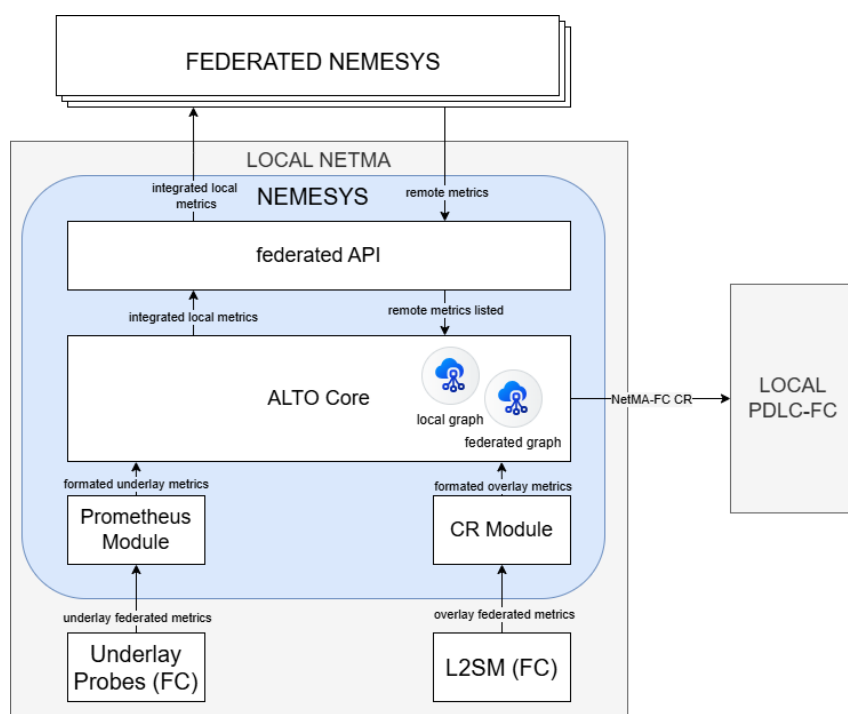


Figure 18: Nemesys-FC functional scheme.

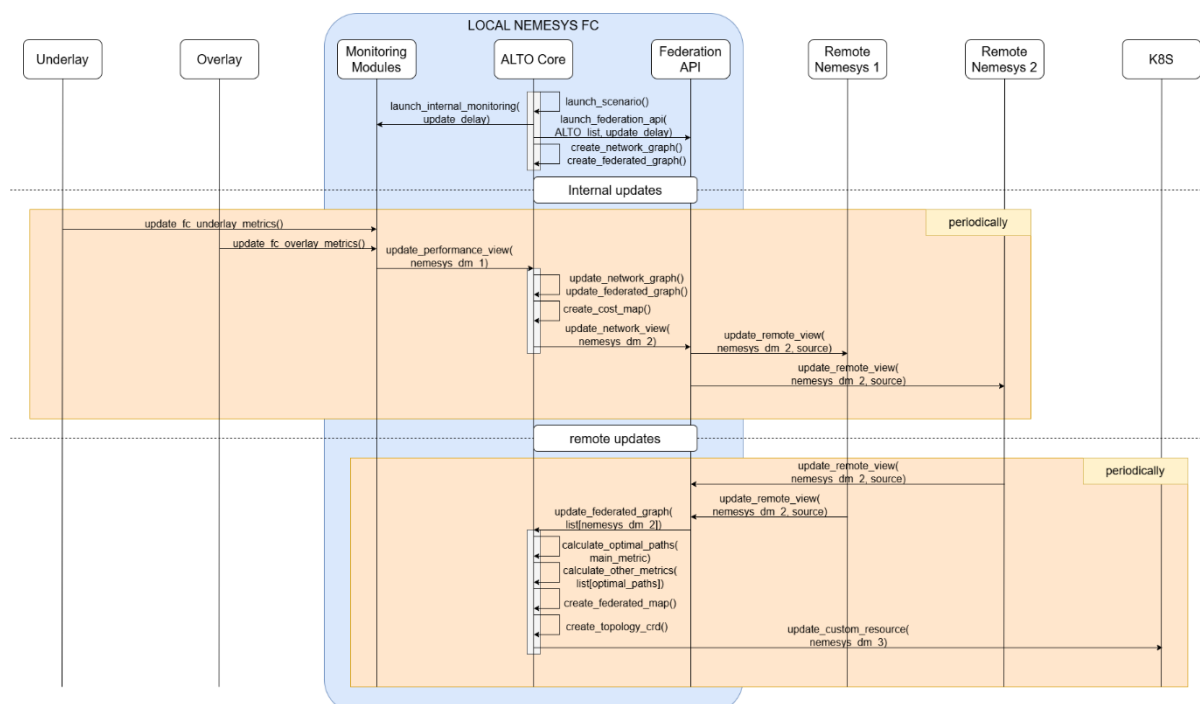


Figure 19: Operation sequence, Nemesis-FC within a MC.

The relationship between internal ALTO Maps and the federated map is illustrated in Figure 20. The federated map is constructed from a collection of one to “N” ALTO Maps, where “N” represents the number of clusters in the federation. Each ALTO Map consists of three sub-maps, generated from a combination of a network map and an overlay graph.

These graphs are built using network data provided by the previously described modules. The data includes a set of nodes and links, along with their associated properties and capabilities, enabling the creation of detailed and context-aware ALTO representations across the federation.

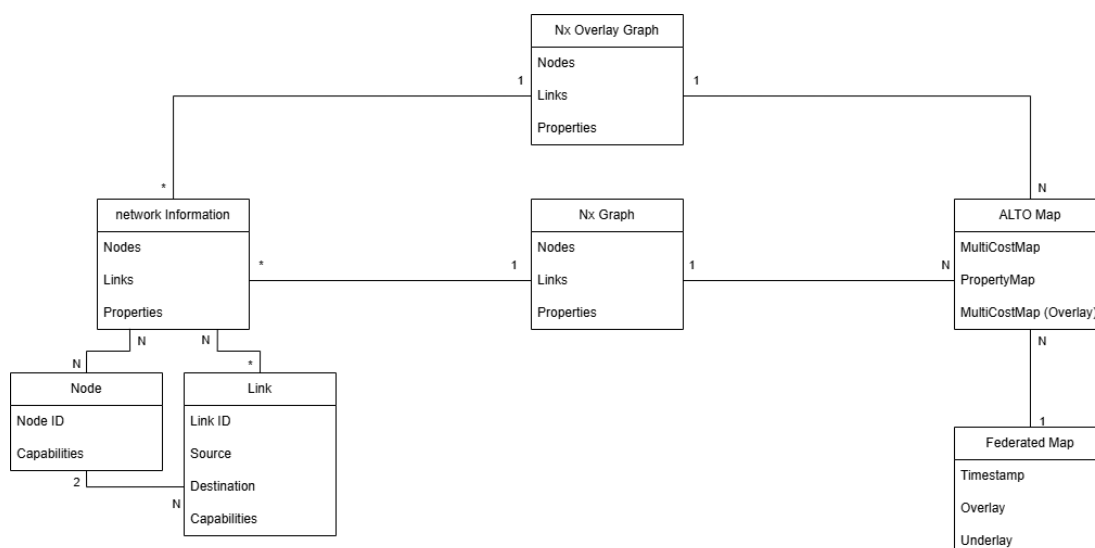


Figure 20: Translation between ALTO maps and CODECO assets, nodes, clusters, links.

3.3.2.3.2 Code Structure

Grant Agreement nr: 101092696

The [Nemesys-FC code](#) is distributed and following the structure defined in the previous section. That is translated into 2 folders with the modules ([modulos](#)) and the APIs ([api](#)) and an additional document with the ALTO Core. Also, there are some additional documents that help as complements.

Table 14: Nemesys-FC main code files.

Dir	File	Function
Nemesys-FC	alto_core.py	Document that contains the main part of the ALTO code, including the main capabilities defined in the RFC7285, but it needs the ampliations to manage the different complements and the functionalities to be deployed, such as the multi-metrics and the multi-graph.
	alto_logger.py	It contains an adapted version for a logger class. To be used for the system to save registers of the events occurred during the code execution. The log files are stored in a folder call logs/, located in the same path as this file.
	config.yaml	Stores configuration settings for the application in YAML format. This file is used to manage environment-specific variables, credentials, or other customizable options outside the codebase.
	LICENSE	States the legal terms under which the project's code can be used, modified, and distributed. In this case is an Apache 2 License (open source).
	Dockerfile	Contains instructions for building a Docker image of the application. It specifies the base image, environment setup, dependencies, and commands to run the application in a containerized environment.
	requirements.txt	Lists Python package dependencies required to run the project.
kuberfiles	kuberfiles/01_netma-fed-topology-crd.yaml	CRD where the multi-cluster topology is defined. It is launched in a deployment phase, and the different instances (the CRs) are created during the running phase.
	kuberfiles/02_nemesys-fc-deployment.yaml	Manifest with the definition of the deployment of the code as well as the different permissions and handlers needed to run the code in a Kubernetes Pod.
modulos	modulos/alto_module.py	Abstracted class used to define how the topology_X.py classes should be structures and the functions to be used to exchange information with the alto_core.py.
	modulos/topology_kubernetes.py	Class that implements the integration between nemesys FC and the probes (overlay

		and underlay) basing their work in Kubernetes and Prometheus tools. It exports the obtained values to the <code>alto_core.py</code> file.
	<code>modulos/topology_ietf.py</code>	Class that implements a topology importer from a file that follows the IETF format. Used to do some tests to validate the topology integration and exchange.
api	<code>api/web/alto_gui.py</code>	Graphic unit to show in an interactive way how the network is connected and the optimal paths.
	<code>api/web/api_http.py</code>	REST API that extends the services defined in the RFC7285. It uses http instead of https by commodity as the pass from HTTP to HTTPS is well defined and therefore it does not bring any new advantage to this PoC code.
	<code>api/web/federation.py</code>	Class that implements a topology importer from a file that follows the IETF format. Used to do some tests to validate the topology integration and exchange.
	<code>api/crd_exporter.py</code>	Document that transforms the ALTO maps into CR updates.
	<code>api/prometheus_exporter.py</code>	Document that transforms the ALTO maps into Prometheus updates

3.3.2.3.3 Dependencies and pre-conditions

There are four types of dependencies to be handled to deploy this component:

- **Hardware dependencies:** There are no hard dependency in terms of capacity or CPUs, being the same requirements as defined to install the CODECO Framework.
- **Software dependencies:** this code requires to have installed Kubernetes, Docker and Python3 installed on the machines to be deployed.
- **Libraries dependencies:** There are some libraries that need to be installed to make this code run. Currently, there is no hard dependency with the versions, but these are the ones that we are currently using (can be updated if needed):
 - `dash==3.0`
 - `kubernetes==31.0`
 - `networkx==3.2`
 - `pandas==2.2`
 - `requests==2.25.1`
- **Configuration dependencies:** Nemesys FC need to have CODECO framework installed, with at least one federated probe (overlay or underlay) available. This component would also need to count with the Kubernetes permissions requested in the [02 nemesys-fc-deployment.yaml](#) manifest.

As this component depends on having other CODECO components installed it would also inherit their dependencies as could be to have the CNI plugins updated, to have access to the sockets or to have a machine with at least 4 virtual cores. These dependencies can vary according with the framework installed therefore this is just a reminder to check also the related dependencies.

Other preconditions are the presence of network performance metrics and having reachability with the other clusters, and being able to identify the services in the inter-cluster management.

3.3.2.3.4 Installation guide

The installation of this [sub-component is automatized](#), requiring the deployment of the necessary dependencies and preconditions, followed by the application of the following kuberfiles:

```
cd nemesys-fc/  
kubectl apply -f kuberfiles/01_netma-fed-topology-crd.yaml  
kubectl apply -f kuberfiles/02_nemesys-fc-deployment.yaml  
cd ..
```

3.3.2.3.5 Interfacing with other components

The Nemesys FC will receive information from other remote Nemesys FC and locally from the network federated probes and the overlay monitoring component.

3.3.2.4 Network Exposure: Nemesys-FC

Nemesys-FC serves as the North-Bound Interface (NBI) within the NetMA component, responsible for gathering and formatting topology and performance data from both overlay and underlay network probes. This processed information is then exposed to other CODECO components, such as the PDLC and ACM, enabling seamless integration and communication across the system. In multi-cluster environments, Nemesys instances must federate by exchanging topological data with their counterparts in each cluster, forming a Peer-to-Peer (P2P) mesh that ensures traceability and a unified network view.

To facilitate this exchange, an ALTO server is implemented, collecting inter-cluster performance metrics and generating cost maps using Dijkstra's algorithm. Each Nemesys periodically shares and receives metrics like latency, bandwidth, packet loss, and energy from its peers, allowing the creation of a comprehensive and consistent network map. The main metric, typically latency, guides optimization decisions, while multi-cost maps allow for a realistic representation of network paths by capturing multiple metrics simultaneously. This federated information is formatted according to the cluster's CustomResourceDefinition and made available to the PDLC for topology and performance management.

3.3.2.4.1 Sequence diagram and Operation Workflow

To support these functionalities, a modular architecture has been adopted for the sub-component. This design includes two dedicated modules responsible for reading information from the overlay and underlay network probes, respectively. Although the overlay and underlay networks coincide—since they reference the same node pairs—the design enables integration of both views into a unified multi-metric graph. As illustrated in Figure 21, this graph structure incorporates multiple performance metrics, with overlay latency selected as the primary metric.

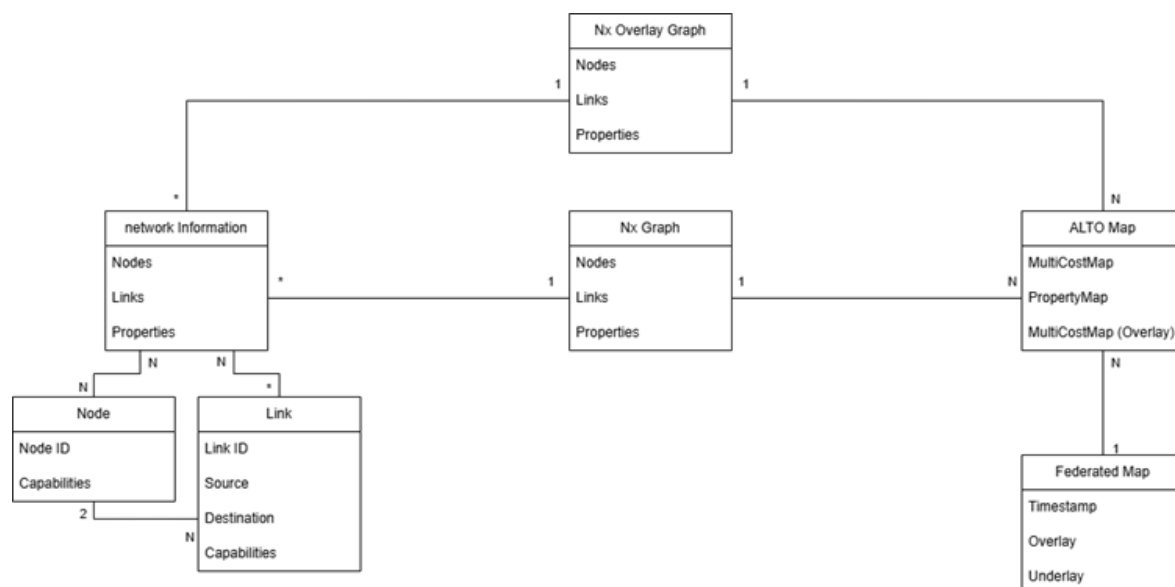


Figure 21: Unified multi-metric graph integrating overlay and underlying network data.

The main module of Nemesys, the ALTO core, will retain this view and create a second one to which it will add the information received from the federation API. It is also responsible for creating the maps that will be exchanged and subsequently formatting the CR.

The ALTO Core also communicates with the federation API to send it the local network map after each update. This map is the information exchanged between the federation members, forwarding just the local views and its updates.

3.3.2.4.2 Architecture and sub-component overview

In order to implement these functionalities, a modular architecture for the sub-component has been followed. In this architecture, there are two modules that will be responsible for reading the information available in the overlay and underlay probes, respectively. Note that the overlay and underlay networks will coincide, since the same node pairs are read, which allows us to integrate both views into the same multi-metric graph.

The ALTO core, module of the Nemesys, will retain this view and create a second one to which it will add the information received from the federation API. It is also responsible for creating maps that will be exchanged and subsequently formatting the CR.

The ALTO Core also communicates with the federation API to send it to the local network map after each update. This map is the information exchanged between the federation members, forwarding just the local views and its updates. The full modules diagram is shown below in Figure 22 and the workflow described is depicted in Figure 23.

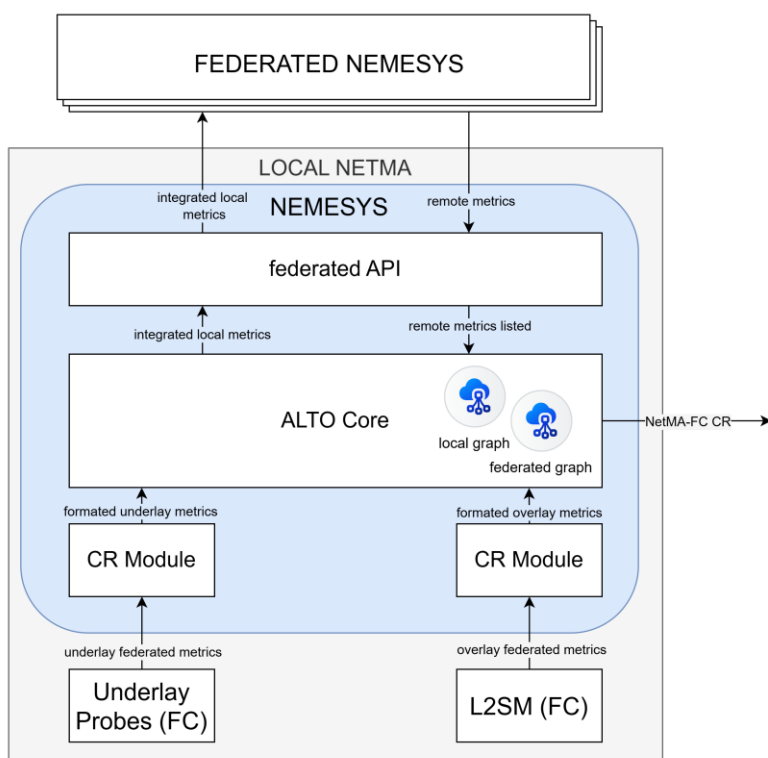


Figure 22: ALTO core interaction with the Federation API, illustrating the creation and exchange of local and federated network maps.

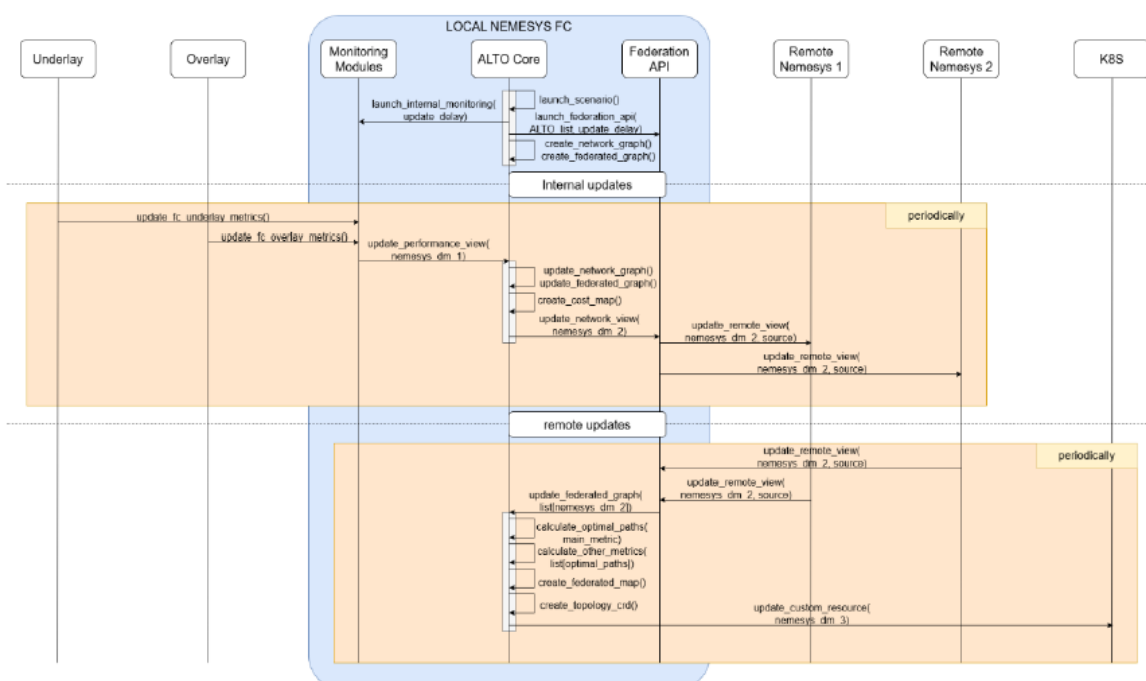


Figure 23: Operation sequence, Nemesis-FC within a MC

The relation between the internal ALTO Maps and the federated ones is described in the Figure 24 , where it is shown that the federated map is made by a set of 1 to “N” ALTO Maps, being “N” the number of clusters in the federation. Each ALTO Map is formed by three maps, which are created using the information from 1 network map and 1 overlay graph. These graphs are composed by network information obtained from the modules described before and that information contains a set of nodes and links with their properties and capabilities.

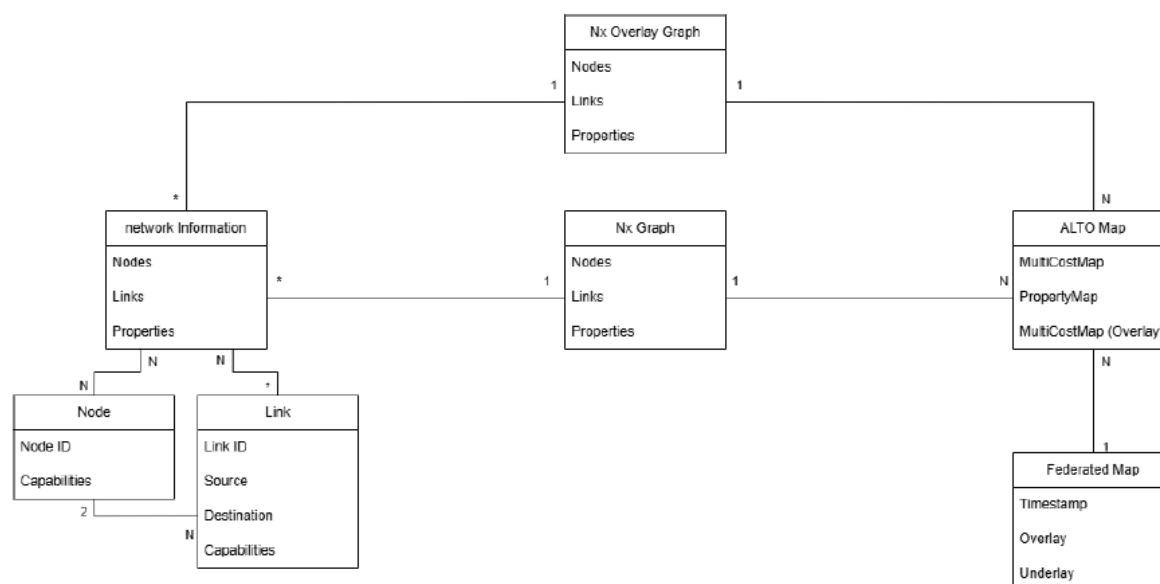


Figure 24: Translation between ALTO maps and CODECO assets, nodes, clusters, links, etc.

3.3.2.4.3 Pre-requisites and installation

Before running the installation, several prerequisites must be met. You need a working Kubernetes cluster that is properly installed and configured to execute kubectl commands. The CODECO framework must already be deployed, as Nemesis-FC depends on its services and architecture. Additionally, at least one federated probe, either overlay or underlay, must be available to monitor and exchange network information. Finally, your cluster must have connectivity with other clusters in the federation, ensuring that cross-cluster communication and coordination can take place correctly.

In order to make a standalone deployment, these steps can be followed:

```
git clone https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/nemesis-fc.git
cd nemesis-fc/
kubectl apply -f kuberfiles/01_netma-fed-topology-crd.yaml
kubectl apply -f kuberfiles/02_nemesis-fc-deployment.yaml
```

MDM-FC: Metadata Manager

3.4 MDM-FC: Metadata Manager

3.4.1 Final Component Description

The CODECO [MDM](#) component collects, links, and enriches metadata related with the applications to be deployed across Edge-Cloud. This metadata assists in better characterizing the application deployment across Edge-Cloud. MDM is therefore a CODECO component that acts as a gateway between the data world (data workflow) and the Kubernetes infrastructure (compute).

MDM and its sub-components are illustrated in Figure 25, simplified using a single Hub cluster and two managed cluster, which may or may not belong to the same neighbourhood. Note that MDM is a distributed system, and its components may be deployed across multiple Kubernetes clusters if configured appropriately. MDM collects metadata from any “native system” that has information on the data, including data stores, catalogues, pipelines that copy and transform data, use-case specific functions that analyse the data, or any other relevant system. For this purpose, MDM relies on **connector interfaces** for each specific data type (I-MDM-E-1). Connectors are deployed on each managed cluster with the API of the corresponding hub cluster where they report to. MDM offers an interface to PDLC (I-MDM-PDLC-1) on the hub cluster which all PDLC instances on the corresponding managed clusters use to gather information about those clusters.

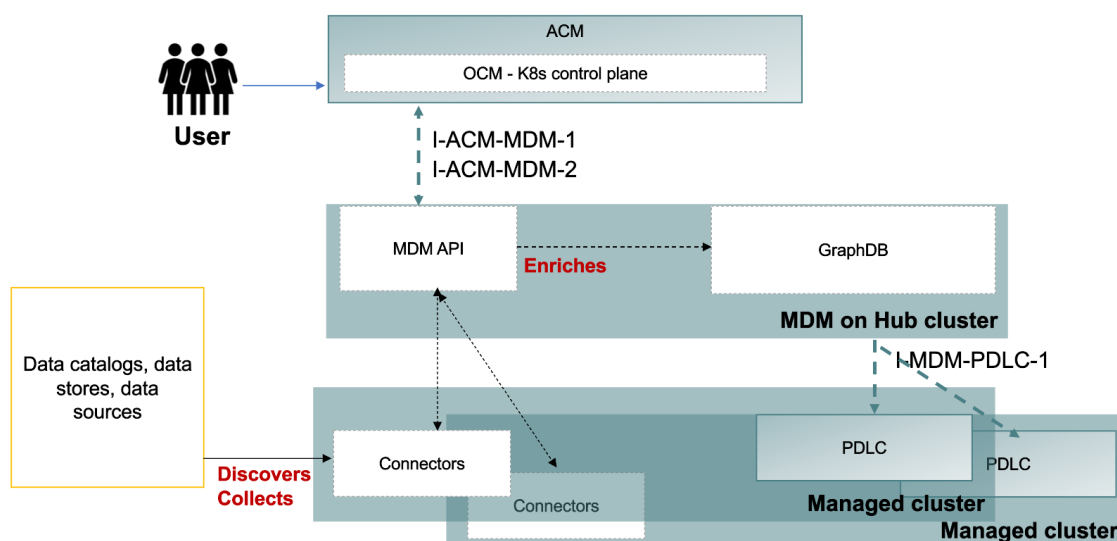


Figure 25: MDM sub-components and APIs to other components

An MDM connector interfaces to a native data system or CRD and pushes metadata into a knowledge graph through the native (internal) MDM API. By adding new connectors or expanding the graph model, the system can be extended to collect any required metadata.

MDM is event-based, relying on Apache Kafka⁵. MDM relies on the Kafka event queue, a log of metadata events that occur for an Edge-Cloud setting managed with CODECO. Events describe changes that have happened and correspond to insert/update/delete of metadata entities and relationships.

⁵ <https://kafka.apache.org/>

MDM materializes (subsets of) the event queue in the knowledge graph to meet the needs of other CODECO components.

MDM is implemented by three components:

- **MDM API:** Implements the REST APIs that allow metadata to be pushed into the graph database and the graph to be queried.
- **Graph Database:** Stores the metadata graph.
- **Connectors:** They gather metadata and push it into the Graph Database using the MDM Controller APIs.

3.4.2 Sub-components

3.4.2.1 Graph Database

The Graph Database ([MDM/Graphdb](#)) is the backend storage of the MDM component. The metadata events from all MDM connectors are consolidated here, making it possible for other components to extract insights from the distributed system from a single pane of glass. It is, of course, possible to request information by directly querying the database using cypher, but other than during development or exploration of the metadata, the MDM component is designed to offer this functionality through the MDM API.

3.4.2.1.1 Code Structure

The main source files of the MDM GraphDB are presented in Table 15.

Table 15: of the main MDM GraphDB source code files.

File	Function
deployment/neo4j-helm.yaml	Example values to deploy the Neo4j Helm chart
Schemas/eu.codecohe/**/*.json	JSON schemas for the entities and relationships in the Knowledge Graph

3.4.2.1.2 Selected Technologies

The graph database is implemented using Neo4j:

- **Neo4j 4.4⁶** or higher.
- **Neo4j APOC⁷:** Awesome Procedures on Cypher (APOC) is an add-on library for Neo4j that provides hundreds of procedures and functions adding a lot of useful functionality.
- **Cypher Graph Query language⁸:** Cypher is a declarative graph query language that allows for expressive and efficient data querying in a property graph. It is the query language for Neo4j and the language opencypher⁹ is based on.

⁶ <https://neo4j.com/>

⁷ <https://neo4j.com/labs/apoc/>

⁸ <https://neo4j.com/developer/cypher>

⁹ <https://opencypher.org/>



- **JSON Schema:** It is used to define the entities, their properties, and relationships among them in a language-independent manner. The schemas are available online¹⁰.

3.4.2.1.3 Pre-requisites

Detailed pre-requisites for running Neo4j in K8s can be found from the vendor directly¹¹. CODECO relies on Neo4j's Community Edition. Commercial utilization of Neo4j's Enterprise Edition may require a license from the vendor. Please see <https://neo4j.com/licensing/> for details.

3.4.2.1.4 Installation Guide

To install the [Graph database](#) refer to the Helm chart provided by Neo4j. The following steps are detailed in the [online documentation in Git](#):

Set the following environment variables:

```
export MDM_NAMESPACE=mdm
export MDM_CONTEXT=docker-desktop
```

Add Neo4j's Helm chart to the Helm repositories:

```
helm repo add neo4j https://helm.neo4j.com/neo4j
```

Update the yaml configuration file for your K8s environment (online version neo4j-helm.yaml)

- Set the neo4j.password for the database user neo4j.
- Define the volumes StorageClass if required.
- Adjust other parameters such as CPU and memory usage as required.

Install the Helm chart:

```
helm --kube-context=$MDM_CONTEXT install mdm-neo4j -n $MDM_NAMESPACE
neo4j/neo4j-standalone -f ./deployment/neo4j-helm.yaml
```

3.4.2.1.5 Inputs & Outputs

The Graph database is fed metadata events from Kafka¹² by the mdm-controller subcomponent and offers a Cypher API to the mdm-api subcomponent. These interfaces are internal to MDM and declared here only for completeness.

3.4.2.2 MDM Controller (APIs)

The MDM controller provides the APIs for other CODECO components to query the metadata graph and for MDM connectors to provide metadata. The MDM Controller is thus a required sub-component for all scenarios where metadata analysis is required for the CODECO use-case. A selected set of MDM connectors depending on the use-case will provide metadata that allows through this subcomponent, allowing other CODECO components like PDLC to

¹⁰ https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb/-/tree/main/schemas/eu.codecohe?ref_type=heads

¹¹ <https://neo4j.com/docs/operations-manual/current/kubernetes/quickstart-standalone/prerequisites/>

¹² <https://kafka.apache.org/>

get summarize information about the systems and data in the form of parameters to models that provide the best scheduling for a given workload.

3.4.2.2.1 Code Structure

Table 16 and Table 17 provide a summary of the main MDM Controller source code files.

Table 16: Description of the main MDM Controller source code files.

File	Function
deployment/*.yaml	Example values to deploy the Kafka, Zookeeper and Neo4j Helm charts
controller/src/helm	Helm chart for the MDM controller
controller/src/python/*.py	Python code for the MDM controller
controller/src/python/ctrl.py	Main program for the MDM controller
controller/Dockerfile	Dockerfile to produce the MDM controller image
controller/requirements.txt	Python dependencies of the MDM controller

Table 17: Description of the main MDM API source code files.

File	Function
mdm-api/src/helm	Helm chart for the MDM API
mdm-api/src/python/*.py	Python code for the MDM API
mdm-api/src/python/app.py	Main program for the MDM API
mdm-api/Dockerfile	Dockerfile to produce the MDM API image
mdm-api/requirements.txt	Python dependencies of the MDM API

3.4.2.2.2 Selected Technologies

The MDM metadata collection has materialized in data systems that make it convenient to manage and exploit the metadata. The key technologies used in the current MDM Controller implementation include:

- **Apache Kafka:** Distributed event store and stream processing.
- **Apache ZooKeeper**¹³, for the overall Kafka management and coordination (e.g., configuration information, naming, providing distributed synchronization).
- **Python 3.9** or higher for the implementation of the APIs and the interaction with Kafka and the Graph database.
- **OpenAPI 3.0** is the standard for the REST API for both input and output.
- **Swagger** is the mechanism to programmatically produce the REST API implementation designs and their documentation.

¹³ <https://zookeeper.apache.org/>

3.4.2.2.3 Pre-requisites

No specific hardware requirements are needed to run the MDM Controller.

3.4.2.2.4 Installation Guide

The Graph database needs to be installed first. After that, the order of installation of the MDM Controller sub-components is important:

- **mdm-zookeeper:** This is the MDM zookeeper sub-component, installed using Helm from the Bitnami¹⁴ chart repository. This is required by the mdm-kafka sub-component.
- **mdm-kafka:** This is the MDM Kafka sub-component, installed using Helm from the Bitnami chart repository. This is required by the mdm-ctrl sub-component.
- **mdm-controller:** The MDM API Controller sub-component. Receives the metadata events from Kafka and pushes them into the Graph database.
- **mdm-api:** Implements the MDM REST API to the Graph database and connector publication API.

Detailed installation information is provided in the [MDM API GitLab repository](#). When developing the MDM subcomponents, the docker build and push description provided in the online documentation require providing the docker registry used to store the images. This, as well as the tag, may change when new versions are developed. Assuming that the mdm-api repository has been cloned locally and is the current directory, the following steps are the basis to install MDM:

Install the MDM controller: If you need to change the tag of the docker image, or any other value, provide it in your own YAML file.

```
cd mdm-controller

helm --kube-context=$MDM_CONTEXT -n $MDM_NAMESPACE install mdm-controller ./mdm-controller/src/helm [-f my_values.yaml]
```

Install the MDM API: If you need to change the tag of the docker image, or any other [value](#), provide it in your own YAML file.

```
cd mdm-api

helm --kube-context=$MDM_CONTEXT -n $MDM_NAMESPACE install mdm-api ./mdm-api/src/helm [-f my_values.yaml]
```

The Helm charts for the installation of the mdm-controller and mdm-api sub-components may become available in the future from an online Helm repository.

3.4.2.2.5 Inputs and Outputs

Events consist of an event envelope and a payload. While the envelope structure is given by MDM, the payload types are application specific. Payloads are either entities or relationships with minimal restrictions on their structure, i.e., they need to contain a type and unique identifier(s). Entities and relationships are defined in JSON schema for CODECO at [MDM/graphdb/schemas/eu.codecohe](#).

¹⁴ <https://github.com/bitnami/charts/tree/main/bitnami>

The MDM API is based on OpenAPI REST (OAS REST API) and integrates the following endpoint groups:

- **Publish** which is the endpoint for connectors writing to the MDM.
- **Events** which are the endpoint to get events from MDM.
- **Graph** gets information by querying the data projection. Initially this would allow for Cypher queries to be performed on the knowledge graph. If, because of the experience gained with partners, the set of queries required for the CODECO operation can be defined in a closed set, they can be incorporated into this API at a later stage.
- **MDM** is the system information endpoint for getting status and resets or other system relevance functions.
- **PDLC** is the endpoint where the PDLC component can query the graph for specific metrics related to the pods in the CODECO application.
- **STORAGE** endpoints get you information about storage classes supported by clusters and storage classes used by Pod Namespace and cluster.

All endpoints are protected and authorized usage enabled. The complete description is in swagger online documentation in MDM at the CODECO Eclipse GitLab (mdm-api/src/python/templates/swagger.yaml). Furthermore, note that the following API groups are not implemented, and responses will always be HTTP status 200 (OK): Groups, Lookup, Registration, Schemas and Versions.

Inputs:

POST /publish/event

Accept in the body a JSON structured event. The event payload object could be an entity or relationship structure.

Outputs:

GET /graph/entity/type

Get a JSON array of existing entity types in the metadata graph.

POST /graph/entity/type/{entitytype}

Get a list of entities in the metadata graph depending on the POST request JSON filter definition.

GET /graph/entity/attributes/{entitytype}

Get a JSON array of all existing attributes in the metadata graph for a given entity type.

GET /graph/path/{edf_id}

Get the graph structure as a JSON object for an entity identified by edf_id. As optional query parameter one can define the number of hops (depth) around the entity that will be included in the resulting graph.

POST /graph/cypher

The POST request body accepts a plain text base CYPHER query. The response will be the default JSON structure produced by the Graph database (Neo4j).

For example:

Request-body:

```
MATCH (a:compute)-[r]-(b) WHERE b.user_id = "Brandy.Neal@example.com"
RETURN a as compute LIMIT 1
```

Response:

```
{
  "identity": 4,
  "labels": [
    "compute",
    "entity",
    "visible"
  ],
  "properties": {
    "_domain": "default",
    "identifier": "AFCF2B98D7",
    "serialNumber": "none",
    "city": "Zurich",
    "ritssDataClassification": "RII unclassified (Non-Confidential)",
    "_status": [],
    "networkClassification": "General Internal Enterprise Network",
    "country_code": "CH",
    "update_time": "2023-04-24T07:54:41.596286000Z",
    "entity_type": "compute",
    "dataClassification": [
      "Public"
    ],
    "name": "srv-22",
    "connector_id": "d79df9ea-efac-360c-b7cc-2cf0de386100",
    "exportClassification": "blue",
    "edf_id": "523e2537-b717-34e0-b303-638b159d7fcb",
    "_id": "523e2537-b717-34e0-b303-638b159d7fcb"
  },
  "elementId": "1"
}
```

Grant Agreement nr: 101092696

GET /mdm

Get the running version of the mdm-api sub-component.

GET /mdm/state

Get the state of the different sub-components (neo4j/kafka/ctrl/etc)

DELETE /mdm/clean

Delete all metadata in the MDM component. Use with caution, only meant for development and testing purposes.

GET /events/sse

Server-Sent Events API-based filtered events from the serialized graph queue in Kafka. This is meant for components that need to be notified of specific metadata events as they happen, as opposed to gathering the current state from querying the metadata graph at time intervals.

GET /pdlc

This endpoint gets the metrics associated with the pods required by PDLC following a CRD format. The parameters are as follows:

- *cluster*: The name of the cluster where the pod is running.
- *namespace*: The namespace where the pod is running
- *pod*: The name of the pod
- *rnd*: (optional) If this is set to "true," the API provides random values for all metrics.

The result is provided in JSON as a K8s CR. The output properties are:

- *pod_name*: The name of the pod, as provided as input.
- *freshness*: How fresh the data processed by pod is (see Prometheus connector below)
- *compliance*: Level of compliance of the pod given where it is running (see Compliance connector below)
- *portability*: A metric of how robust the execution of the pod is where it is running (see K8s connector below).

GET /storage/classes

This endpoint gets supported storage classes. The parameter as follows:

- *cluster*: The name of the cluster where the pod is running.

The result is provided in JSON array of strings as storageclasses name.



```
GET /storage/pvc
```

This endpoint gets used storageclasses by pod. The parameter as follows:

- *cluster*: The name of the cluster where the pod is running.
- *namespace*: The namespace where the pod is running
- *pod*: The name of the pod

The result is provided in JSON array with `pvc_name` and `storage_class_name`.

e.g.: [{"pvc_name": "data-mdm-neo4j-0", "storage_class_name": "gp2"}]

3.4.2.2.6 UML Diagram

The momentary interaction between the CODECO components and MDM and among the MDM subcomponents is depicted in Figure 26.

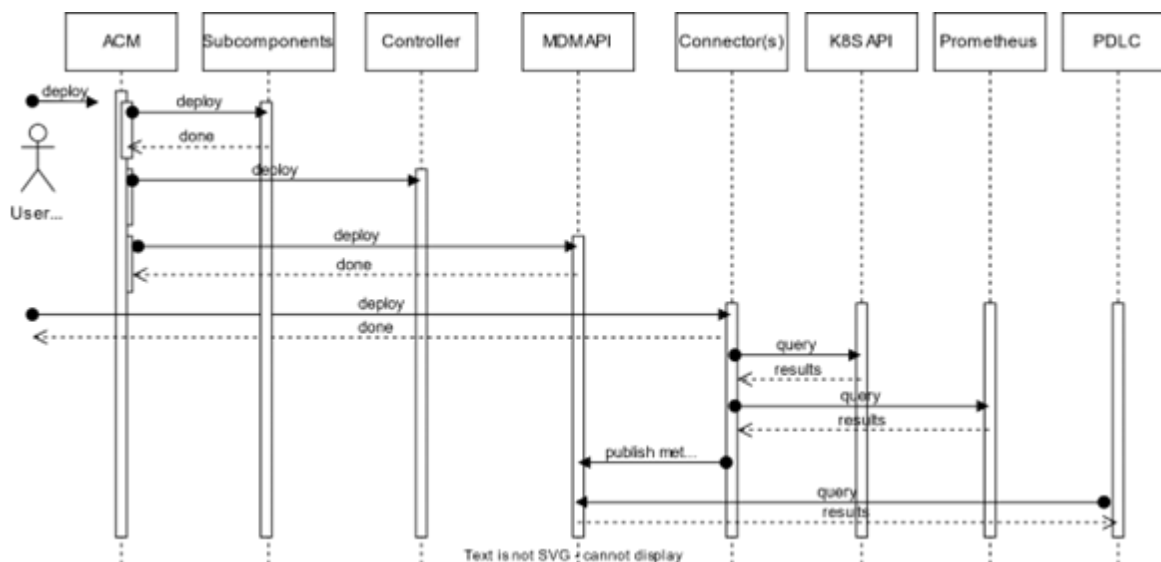


Figure 26: Interaction between MDM subcomponents.

Note that the connectors interact with the K8s API and the Prometheus API to obtain metadata, and the PDLC component uses the MDM API to obtain the metrics obtained from that metadata. As explained in Figure 5, also note that this interaction may happen inside a Kubernetes cluster or across clusters.

3.4.2.3 Connectors

Connectors send metadata to MDM in the form of events. Events are structured JSON documents. An event contains the following elements:

- The event type, either “insert” or “delete.”
- An identifier that uniquely identifies the connector that issued the event.
- A timestamp.
- The payload, i.e., the metadata.

Metadata is sent as entities and relationships. MDM imposes a basic structure on both entities and relationships to ensure that metadata can be stored as a graph. Entities must have a globally unique identifier and a type. It is the task of the connector to assign these. In addition, entities contain arbitrary numbers of attributes. The mandatory elements of a relationship are source and target entity identifier as well as a type. Relationships do not carry attributes.

MDM does not define or enforce a static data model. Instead, the graph data model is defined by the structure of the entities and relationships that are inserted into MDM.

Entities and relationships are defined in JSON schema for CODECO at [MDM/graphdb/schemas/eu.codecohe](https://mdm.graphdb.schemas.eu.codecohe).

3.4.2.3.1 Selected Technologies

Connectors may be implemented in any programming language, if they make metadata available following the data model of the application framework (in this case CODECO's) and use the MDM Controller API conventions to make it available to the Graph database.

In the context of CODECO, the following technologies are considered:

- **Python 3.9 or higher** is the preferred programming language, as it will allow the connector developer to leverage the open-source connector SDK.
- **IBM Pathfinder SDK:** Available here¹⁵, it provides YAML-based configuration, and the base communication means for the connector to send metadata events to the MDM API.

3.4.2.3.2 Pre-requisites

MDM Connectors do not have any hardware requirements. The MDM Controller component needs to be installed first so that connectors can send the metadata events.

3.4.2.3.3 Installation Guide

MDM Connectors are installed using Helm charts. The parameters required depend on the source of metadata to be inspected by the connector and thus cannot be listed extensively.

At a minimum, the following parameters need to be provided:

- **pathfinder.url:** The URL of the MDM API. This may be a single string, or an array containing multiple MDM instances to report to, when the neighbourhoods span multiple hub clusters.
- **pathfinder.K8sUrl:** The Kube API of the K8s cluster where the connector runs.

For a complete list of parameters please refer to pathfinder-python-sdk/ibm_pathfinder_sdk/pathfinderconfig.py.

3.4.2.3.4 Inputs & Outputs

MDM Connectors get input from the source of metadata inspected. The output is always provided in the form of metadata events sent to the MDM API using the **/publish/event** POST request.

For CODECO, helper classes are provided for Python connectors in connectors/metadata-model/src/main/python/. This way, connector developers can use the MDM CODECO metadata model directly by instantiating these Python classes for entities and relationships.

¹⁵ <https://pypi.org/project/ibm-pathfinder-sdk>

3.4.2.4 CODECO Default connectors

Users of the CODECO framework can install any connectors they develop or use the MDM APIs to provide metadata into the system to compute metrics or other information for their own purposes. Out of the box, CODECO provides three connectors that provide the metadata for certain metrics that are used by the scheduling mechanism in PDLC to decide on the placement of the CODECO application. These connectors are described in the following sections.

3.4.2.4.1 K8s connector

Information about the K8s environment managed by CODECO is essential to provide meaningful decisions as to how to schedule CODECO applications. The K8s connector gets all entities in a K8s cluster and includes them in the MDM metadata graph, together with the relationships among them. The K8s connector collects among other data the number of restarts of a pod, which is used as a proxy for the portability metric of that pod running in that cluster and namespace.

3.4.2.4.1.1 Code Structure

Table 18 provides a summary of the main MDM K8s connector source code files.

Table 18: Main MDM K8s connector source code files.

File	Function
connectors/k8s/src/main/helm	Helm chart for the MDM K8sconnector
connectors/k8s/src/main/python/*.py	Python code for the MDM K8sconnector
connectors/k8s/src/main/python/connector.py	Main program for the MDM K8sconnector
connectors/k8s/src/main/docker/Dockerfile	Dockerfile to produce the MDM K8sconnector
connectors/k8s/src/main/python/requirements.txt	Python dependencies of the MDM K8sconnector

3.4.2.4.1.2 Selected Technologies

The K8s connector is developed based on:

- **Python 3.9 or higher:** Python is the preferred programming language, as it will allow the connector developer to leverage the open-source connector SDK.
- **IBM Pathfinder SDK:** Available online¹⁶, it provides YAML-based configuration, and the base communication means for the connector to send metadata events to MDM's API.
- **K8s watcher API:** This allows the K8s connector to receive as events all the entities created in the K8s cluster, including but not limited to pods, CRDs, deployments, etc.
- The K8s connector source code is available at [MDM/connectors/connectors/k8s/src/main](#).

3.4.2.4.1.3 Pre-requisites

The CODECO MDM K8s connector needs to run with enough privilege to get all events in a K8s cluster.

¹⁶ <https://pypi.org/project/ibm-pathfinder-sdk>

3.4.2.4.1.4 Installation Guide

The CODECO MDM K8s connector is installed using a Helm chart provided in the same repository where the code is available. Other than the default parameters required for all connectors, the following are required:

- **crawler.K8s.api_url**: The URL of the K8s API for the cluster being monitored.
- **crawler.K8s.sa_token**: If the connector is not running under a sufficiently privileged service account, the token with sufficient permissions to watch the K8s API must be provided in this parameter.

3.4.2.4.1.5 Inputs and Outputs

Using as input the events obtained by watching the K8s API, the K8s connector provides metadata about all K8s entities in the MDM graph following the metadata model described above.

3.4.2.4.2 Compliance connector

The level of compliance of a K8s environment managed by CODECO is another metric required to schedule CODECO applications. The Compliance connector uses Kubescape¹⁸ to obtain a metric of compliance of the cluster and adds that information to the MDM metadata graph.

3.4.2.4.2.1 Code structure

Table 1919 provides a summary of the main MDM Compliance connector source code files.

Table 19: Description of the main MDM Compliance connector source code files.

File	Function
connectors/kubescape/src/main/helm	Helm chart for the MDM compliance connector
connectors/kubescape/src/main/python/*.py	Python code for the MDM compliance connector
connectors/kubescape/src/main/python/connector.py	Main program for the MDM compliance connector
connectors/kubescape/src/main/docker/Dockerfile	Dockerfile to produce the MDM compliance connector
connectors/kubescape/src/main/python/requirements.txt	Python dependencies of the MDM compliance connector

3.4.2.4.2.2 Selected Technologies

The K8s connector is developed based on:

- **Python 3.9 or higher** is the preferred programming language, as it will allow the connector developer to leverage the open-source connector SDK.
- **IBM Pathfinder SDK**: Available online¹⁷, it provides YAML-based configuration, and the base communication means for the connector to send metadata events to MDM's API.

¹⁷ <https://pypi.org/project/ibm-pathfinder-sdk>

- **Kubescape:** Kubescape is an open-source technology that performs several tests (controls) following compliance rules from well-known security and compliance frameworks. By default, the MITRE and NSA frameworks are enabled¹⁸, which perform controls on configuration of worker nodes, K8s API, etc., and provide a level of coverage as a percentage.

3.4.2.4.2.3 Pre-requisites

The CODECO MDM K8s connector needs to run with enough privilege to access configuration information through the K8s API.

3.4.2.4.2.4 Installation Guide

The CODECO MDM Compliance connector is installed using a Helm chart provided in the same repository where the code is available. Other than the default parameters required for all connectors, the following are required:

- **crawler.K8s.api_url:** The URL of the K8s API for the cluster being monitored.
- **crawler.K8s.sa_token:** If the connector is not running under a sufficiently privileged service account, the token with sufficient permissions to obtain configuration information through the K8s API must be provided in this parameter.

3.4.2.4.2.5 Inputs and Outputs

Using as input the information available through the K8s API, the Compliance connector uses the Kubescape tool to provide a metric for the level of compliance of a K8s cluster and the worker nodes, which is attached to the corresponding MDM metadata graph entities.

3.4.2.4.3 Prometheus connector

CODECO components and use-cases may use Prometheus to make certain metrics available. The MDM Prometheus connector obtains configured metrics from Prometheus to enrich entities in the metadata graph with those values. In particular, the “freshness” metric is captured such that if a CODECO application is to be scheduled according to that metric, making it available in CODECO’s Prometheus will make it possible for PDLC to schedule the application to optimize the “freshness” of the data being processed.

3.4.2.4.3.1 Code Structure

Table 20 provides a summary of the main MDM Compliance connector source code files.

Table 20: Description of the main MDM K8s connector source code files.

File	Function
connectors/prometheus/src/main/helm	Helm chart for the MDM prometheus connector
connectors/prometheus/src/main/python/*.py	Python code for the MDM prometheus connector
connectors/prometheus/src/main/python/connector.py	Main program for the MDM prometheus connector

¹⁸ <https://kubescape.io/docs/frameworks-and-controls/frameworks/>

connectors/prometheus/src/main/docker/Dockerfile	Dockerfile to produce the MDM prometheus connector
connectors/prometheus/src/main/python/requirements.txt	Python dependencies of the MDM prometheus connector

3.4.2.4.3.2 Selected Technologies

The K8s connector is developed based on:

- **Python 3.9 or higher** is the preferred programming language, as it will allow the connector developer to leverage the open-source connector SDK.
- **IBM Pathfinder SDK**¹⁹ provides YAML-based configuration and the base communication means for the connector to send metadata events to the MDM API.
- **Prometheus**: Prometheus is an open-source technology²⁰ that records metrics in a time series database built using a pull model, where an endpoint is made available for applications and Prometheus queries periodically the endpoint to obtain the metrics.

3.4.2.4.3.3 Pre-requisites

The CODECO MDM K8s connector needs the required credentials to access the Prometheus query endpoint. The Prometheus service needs to be integrated with K8s so that the association between the metric and the pod producing it can be obtained and reflected in the Knowledge Graph.

3.4.2.4.3.4 Installation Guide

The CODECO MDM Compliance connector is installed using a Helm chart provided in the same repository where the code is available. Other than the default parameters required for all connectors, the following are required:

- **crawler.k8s.api_url**: The URL of the K8s API for the cluster being monitored.
- **crawler.prometheus.server.url**: The hostname of the Prometheus query API
- **crawler.prometheus.server.port**: The port of the Prometheus query API
- **crawler.prometheus.metric**: The name of the metric to capture (default is *CODECO_freshness*)
- **crawler.prometheus.entity**: The name of the entity in the metadata graph to attach the metric to.

3.4.2.4.3.5 Inputs and Outputs

Using as input the values obtained through the Prometheus query API; an arbitrary metric can be attached to arbitrary entities in the MDM metadata graph.

3.5 SWM-FC: Scheduling and Workload Migration

This section extends the description of [SWM](#). In the following “SMW” refers to SWM for the single-cluster case and “SWM-FC” to SWM for the multi-cluster case. The section contains the aspects that are *different* between SWM and SWM-FC, i.e., most of the definitions, terms,

¹⁹ <https://pypi.org/project/ibm-pathfinder-sdk/>

²⁰ <https://prometheus.io>



and components of SWM apply to SWM-FC as well. Their description is not repeated here for brevity's sake (see D10 for further information).

Figure 27 shows the environment SWM-FC is working in. There is a set of clusters C_1 , C_2 , and C_3 . Each cluster has some nodes (e.g., C_1 has nodes N_1 , N_2 , and N_3) connected by a network. With the L2SM network provided by NetMA, one of the nodes (e.g., N_1 in C_1) is a so-called network edge device (NED) that connects clusters. SWM-FC does not require the network connecting clusters to be fully connected. In the figure below there is no connection between C_2 and C_3 .

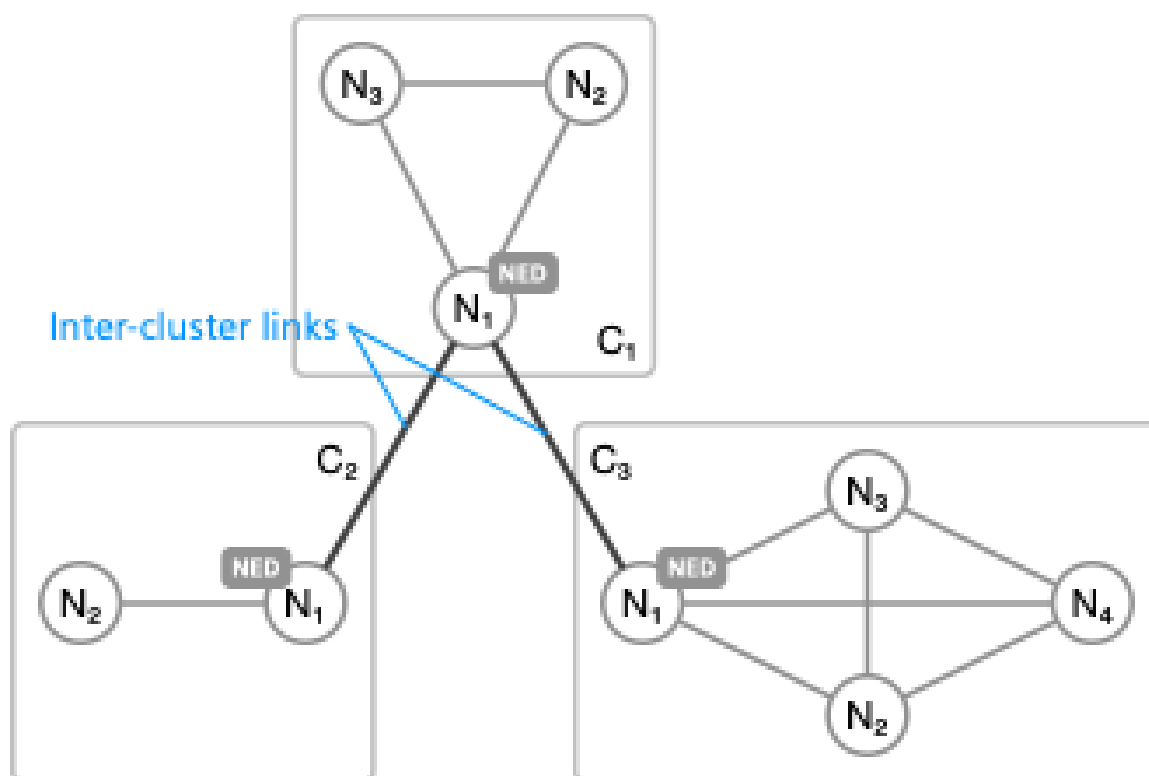


Figure 27: A neighborhood with three clusters connected by an L2SM network

SWM-FC supports multiple networks (with different service classes) between clusters. SWM-FC uses the networks' names to connect networks of different clusters. If network "ls2m" is available in clusters C_1 and C_3 and there is a network "l2sm" on the cluster level, then clusters C_1 and C_3 are connected via network "l2sm". SWM-FC places two workloads connected by a channel either in a single cluster or in two connected clusters.

3.5.1 Component Description

SWM-FC has the same task as SWM: place workloads on nodes and establish network connections between those workloads, as represented in Figure 28. The difference is that SWM-FC may now place workloads in one or more clusters as it sees fit. SWM-FC assumes that an application's workloads may be placed in different clusters and that applications in a group may be put into different clusters.

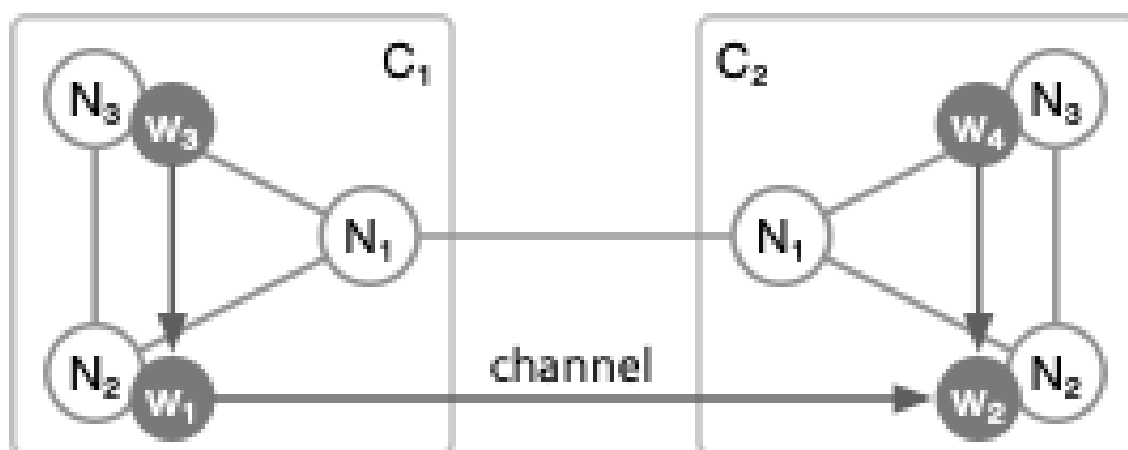


Figure 28: An application with workloads in two clusters.

SWM-FC is concerned only with workloads and channels. It creates pods for workloads and establishes paths in suitable networks for the specified channels (with respect to a channels requirement). An application in Kubernetes may refer to a lot more objects (e.g., ConfigMaps, Secrets, Volumes). These objects are not part of SWM-FC's application model. SWM-FC assumes that all these "secondary" objects are appropriately configured before an application is started or created on the fly based on the assignment plan.

SWM-FC supports placing a group of applications in a set of clusters considering the requirements specified (CPU, memory, latency, bandwidth...). The initial placement can now involve multiple clusters. An application's workloads are not required to be all in the same cluster. Workload migration may be done between clusters. SWM-FC maintains the application's requirements also in the multi-cluster case, i.e., a migration is done only, if SWM-FC finds a suitable placement that does not violate any constraints.

Workload migration with SWM-FC is similar to the single-cluster case. SWM-FC migrates the workloads (i.e., the pods) and the corresponding network connections, but it does *not* take care of the state within a pod. This is consistent with how Kubernetes handles pods.

3.5.1.1 SWM-FC Application Model

A user or client like ACM-FC interacts with SWM-FC by specifying the desired state in custom resources (CRs). The custom resource definitions (CRDs) for SWM-FC are an extended version of those of SWM. There is only one new resource type with information about a cluster. It defines an endpoint for a cluster and a polling interval for updating the cluster's status. A cluster's status indicates whether a cluster can be reached and how many federated workloads it hosts. SWM-FC does not use any other information when scheduling in a federated environment. In particular a cluster's infrastructure is private to the cluster and even node or links names do not leave the cluster.

Clusters that SWM-FC should use for scheduling must be connected. SWM-FC uses CRs of type NetworkTopology for the links between clusters. There may be one network per service class connecting clusters (i.e., one best-effort network and one assured network). If clusters A and B are not connected by a network for instance of service class "assured", it is not possible to set up an assured channel between a workload in cluster A and a workload in cluster B.

The network between clusters has a restriction that networks within a cluster do not have: The network does not support paths with more than one link. This means that two clusters are connected only if there is direct link between them. If clusters A, B, and C are connected by

links $A \leftrightarrow B$ and $A \leftrightarrow C$, no channel can be built between clusters B and C. The network path consisting of the links $A \leftrightarrow B$ and $A \leftrightarrow C$ is *not* used for traffic between B and C.

Most new fields of SWM-FC's resources should be used with care to give SWM-FC maximum flexibility to place workloads. An application for instance has a field `cluster` that fixes an application to a particular cluster. SWM-FC respects this setting but may not be able to come up with a placement in this case.

If a user has a set of resources for some applications for a single cluster, it is sufficient to add a list of target clusters to an application group to indicate that SWM-FC should place the applications in those clusters, i.e., the applications should run in a multi-cluster environment.

3.5.1.2 SWM-FC Instances

SWM-FC aligns with the CODECO view of cluster management that is the one from OCM. There are two kinds of clusters: hub clusters and managed clusters. A hub cluster is used for managing a set of clusters. Only the managed clusters are used to run workloads of a CODECO application. The hub cluster controls its set of clusters (e.g., install software).

SWM-FC runs in each cluster, i.e., one instance is running in the hub cluster and one instance is running in each managed cluster. Each instance can act as an SWM instance for its cluster. SWM-FC can only access detailed information about a cluster when it is running within that cluster, i.e., a cluster's internal structure is not visible outside the cluster. This is due to privacy and scaling reasons.

While the installed software is identical, the roles of the two instances in SWM-FC are different. Only the instance in the hub cluster handles requests to place a group in a set of clusters. The instances in the managed clusters forward any multi-cluster request to the hub cluster and place workloads within their cluster only. The instance in the hub cluster orchestrates the actions of the instances in the managed clusters when placing a group in a set of clusters.

3.5.1.3 Federated Scheduling with SWM-FC

SWM-FC places a new group in a set of clusters in three steps. If any of the steps fails, no placement is found and SWM-FC tries again later. Only the final step actually allocates cluster resources. The target set of clusters may not be changed once a group has started placing workloads in clusters. The set of target clusters for a group is called a neighbourhood. Different groups can have different neighbourhoods.

1. Order the neighbourhood's clusters.
2. Assign workloads within the target clusters, cluster by cluster.
3. Implement the place in the target clusters.

Within the hub cluster SWM-FC uses an /L1 problem/ to track a placement's progress. An L1 problem tracks which workload reside in which cluster, but it is *not* known on which node within the cluster. SWM-FC in the hub cluster does not *solve* an L1 problem. The instances of SWM-FC in the managed clusters build up the assignment of workloads to clusters. The instance in the hub cluster just tracks what is where to see whether the placement is done.

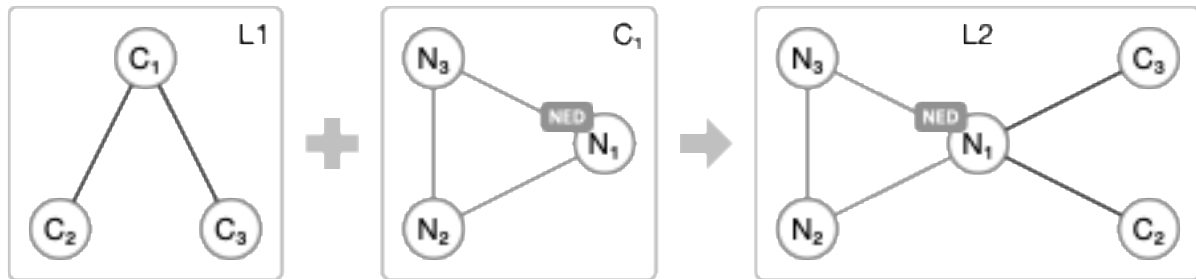


Figure 29: $L1$ problem and infrastructure of cluster C_1 yield the $L2$ problem to solve in C_1

An $L1$ problem, as represented in Figure 29, consists of the set of applications to be deployed and the infrastructure. For an $L1$ problem the infrastructure consists of the neighbourhood's clusters and the links between them as defined by the inter-cluster networks. SWM-FC uses only clusters that are "reachable", i.e., the cluster's SWM-FC instance responded to a PING request.

SWM-FC places workloads within target clusters considering the other clusters of a neighbourhood (a so-called $L2$ problem). An $L2$ problem for SWM-FC is similar to a problem that SWM handles but it needs to take workloads in other clusters into account. In an $L2$ problem a cluster is represented by a special node. As there is intentionally no information about a cluster's resources available, a cluster node has "infinite" resources and a very fast network as far as scheduling is concerned.

Step 1 puts a neighbourhood's clusters in a suitable order for placing the workloads. The order for clusters in S depends on

- the number of node recommendations for a cluster,
- the number of workloads running in a cluster, and
- the latency of inter-cluster links.

Considering these values ensures that a cluster that already has a large number of recommendations or workloads is tried first to avoid inter-cluster communication. The last condition ensures that faster links between clusters are tried first.

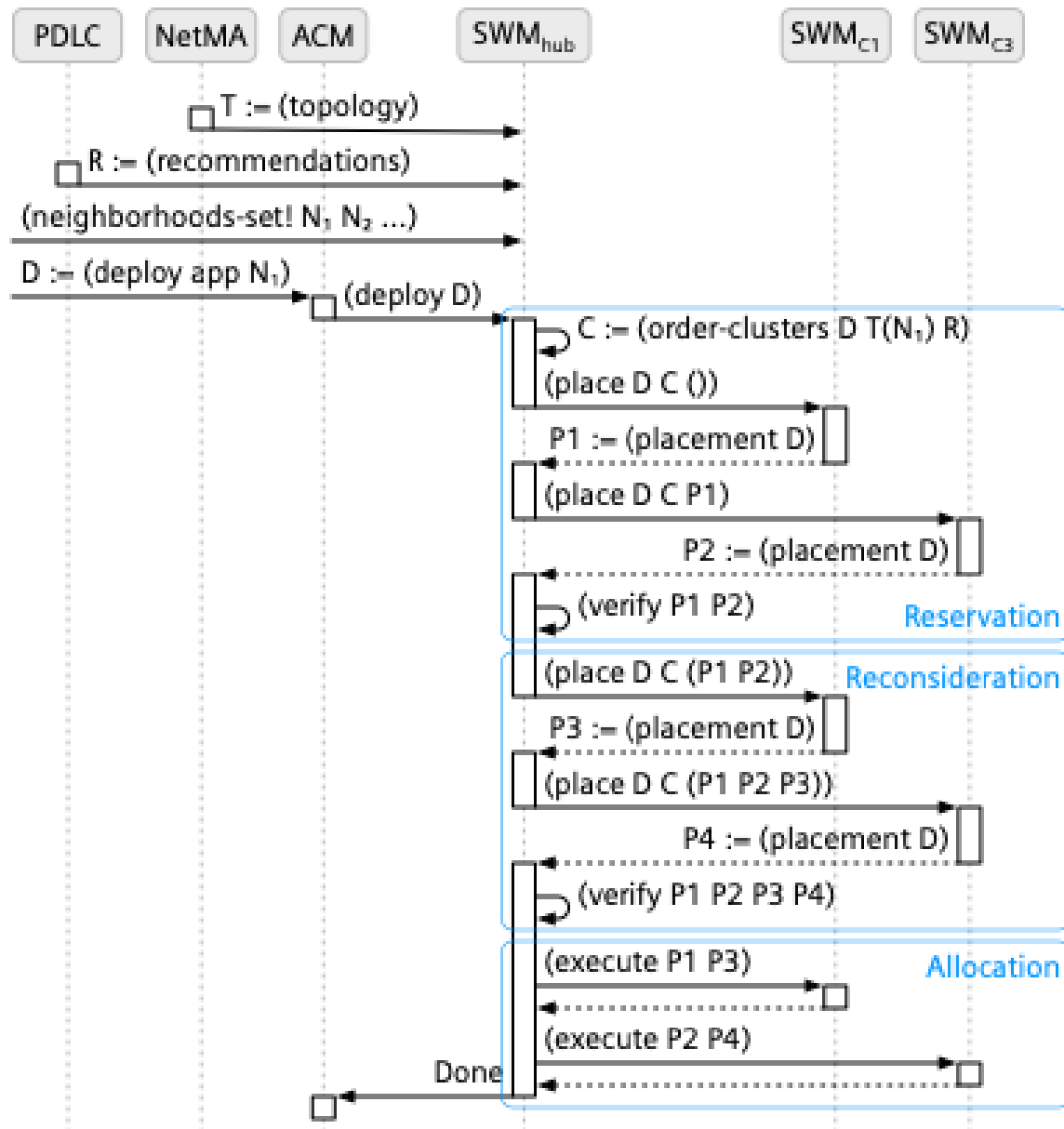


Figure 30: Event-sequence diagram for placement of an application.

Step 2 assigns workloads to nodes in a managed cluster, i.e., it solves an L2 problem. The first cluster assigns greedily all workloads it can support considering any node recommendations and assigns all other workloads tentatively to other managed clusters. A workload placement solver can assign any number of workloads and channels to a cluster without violating resource constraints. When the L2 problem of the particular cluster is solved later, actual resources are checked. SWM-FC does not assign workloads to clusters for which a placement was already done.

The SWM-FC instance running in a managed cluster sends its solution back to SWM-FC running in the hub cluster. This instance modifies the placement problem (an L1 problem) to include the solution from the managed cluster and asks the next managed cluster to solve the enhanced problem. If all workloads are assigned, the placement is complete and can be executed (i.e., step 3 gets executed).

It is not a problem if SWM-FC cannot place a single workload in a managed cluster. The set of workloads does not shrink, but the number of clusters to visit is smaller. If all clusters were visited and there are still some workloads to place, this is a failure unless SWM-FC goes for increased robustness. The event sequence diagram shows this phase as “Reconsideration”. Running this second phase is the default but this can be configured when SWM is deployed.

Assigning channels between workloads running in the same cluster is easy. SWM-FC can do this in the way, as SWM. Channels connecting workloads in different clusters are harder. Consider a channel connecting workloads in cluster C_1 and C_2 . As information about cluster C_2 is not available when solving the L2 problem for cluster C_1 , SWM-FC can only check whether the part of the channel in the cluster C_1 has no problems. The final check for a channel's bandwidth and latency is done when SWM-FC places the second workload in cluster C_2 .

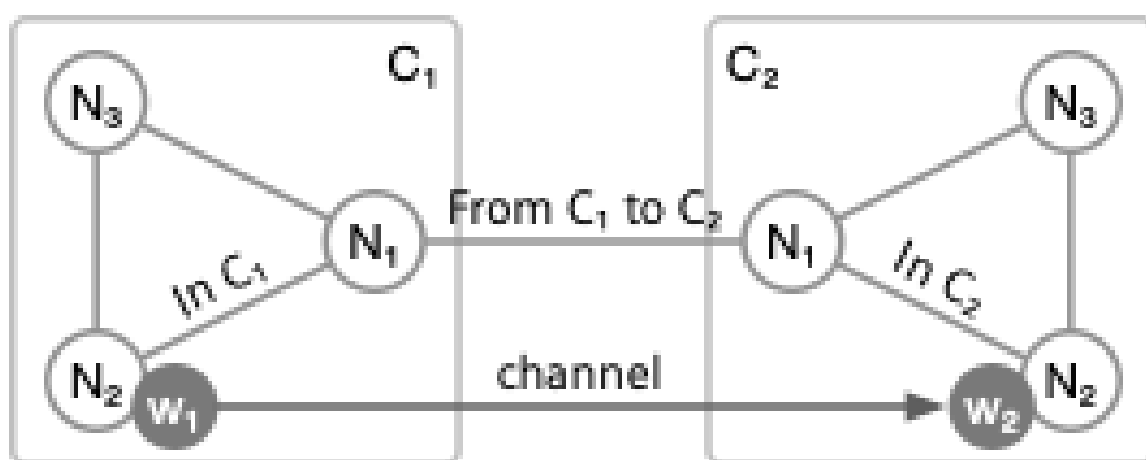


Figure 31: A channel connecting workloads into two clusters has three parts as far as the latency is concerned.

For this check, cluster C_2 must know the channel's latency in cluster C_1 . The total latency of the channels is the latency in cluster C_1 , the latency of the link connecting clusters C_1 and C_2 , and the latency in cluster C_2 . SWM-FC can guarantee that the channels' requirements are met. Note that this works even if cluster C_2 is changed to cluster C_3 during the scheduling process for some reason. The latency in cluster C_1 is added to the specification of the placement.

SWM-FC tries to minimize the latency in a channel's first cluster (i.e., cluster C_1 in this case). This increases the flexibility at the other end. Increased flexibility is important especially if a workload must be moved to another cluster (e.g., cluster C_3) which might have a different inter- and inner-cluster latency.

If SWM-FC can solve all L2 problems, it is guaranteed that the final solution satisfies the requirements of all workloads, channels within clusters, and also channels between clusters.

To increase robustness, SWM-FC reconsiders the ordered set of clusters S when it could not assign all workloads. For all workloads that could not be assigned, SWM-FC relaxes the placement restrictions and asks all clusters in S in the same order to assign the remaining workloads. If this second round does not place all workloads, the placement fails.

Executing a second round of placement requests in S is not as expensive as it seems at first glance. The information on all problems is already available and the problem to solve is smaller. The instances in the managed cluster use the same infrastructure information as for the first round.

When all L2 problems are solved, the placement found must be implemented in the different clusters. While solving L2 problems, cluster resources are *not* reserved. The final step to set up communication channels in the corresponding networks and create pods for workloads. The hub cluster establishes any links with the inter-cluster networks used by the placement. If one of the implementation steps fails, all allocations (i.e., in all managed clusters) are reverted and the placement fails.

For channels connecting workloads in two clusters, the reservation for the part between clusters is managed by the instance in the hub cluster. This instance must have the privileges to control the networks between clusters. For the L2SM networks provided by NetMA, the hub cluster's NetMA instance is asked to establish the links between clusters.

SWM-FC handles workload migration in a similar way. PDLC triggers migration by adding or modifying node recommendations to workloads in an application's CR. PDLC modifies the application within a managed cluster. SWM-FC forwards the node recommendation appropriately to the instance in the hub cluster that triggers a reconciliation for this change. The change is not forward "as-is" since it contains node names that may not leave the cluster.

SWM-FC executes the same three steps, but the final step is a bit more complex. SWM-FC reserves resources for any migrated workload *before* the old instance is stopped. This requires more resources but allows to switch from old to new instances swiftly. It also avoids problems with a service's availability when an old instance is released early but a new instance to replace it cannot be created for some reason.

3.5.1.3.1 Scheduling Example

This section shows how to place application app with two workloads w1 and w2 in a neighbourhood having three clusters C1, C2, and C3. For brevity's sake the example has only a single network. The language used to show the problem is the workload placement solver's text format for assignment problems.

```
workload assignment v2
model example
application app
workload w1,w2 needs cpu:500 ram:1GiB
channel c12:w1->w2 needs lat:10ms bw:10Mb
```

The neighbourhood is defined in the resource of type `ApplicationGroup`. SWM-FC operates on groups that can contain more than one application. A group is treated as a unit, i.e., all its applications are placed in the federated environment.

```
apiVersion: qos-scheduler.siemens.com/v1alpha1
kind: ApplicationGroup
metadata:
  name: example
  namespace: default
spec:
  minMember: 1
  neighbourhood:
```


Grant Agreement nr: 101092696

- C1
- C2
- C3

The clusters in the neighbourhood are identical. They have three worker nodes N1, N2, and N3 connected by an L2SM network. Node N1 in each cluster is the so-called network edge device (NED). All traffic to other clusters goes through the NED. The links in the inter-cluster network are the links between NEDs.

```

infrastructure
node N1 provides ram:2GiB cpu:400 // NED
node N2 provides ram:3GiB cpu:900
node N3 provides ram:2GiB cpu:400

network l2sm
link l12,l21:N1<->N2, l13,l31:N1<->N3, l23,l32:N2<->N3
    with lat:1ms bw:100Mb
paths (p12:l12 p21:l21 p13:l13 p31:l31 p23:l23 p32:l32)

```

Two links form the inter-cluster network between clusters C1, C2, and C3. This network does *not* connect every cluster to every other cluster. For making a placement, it is not necessary for the graph of the inter-cluster network to be a complete graph, as represented in Figure 32.

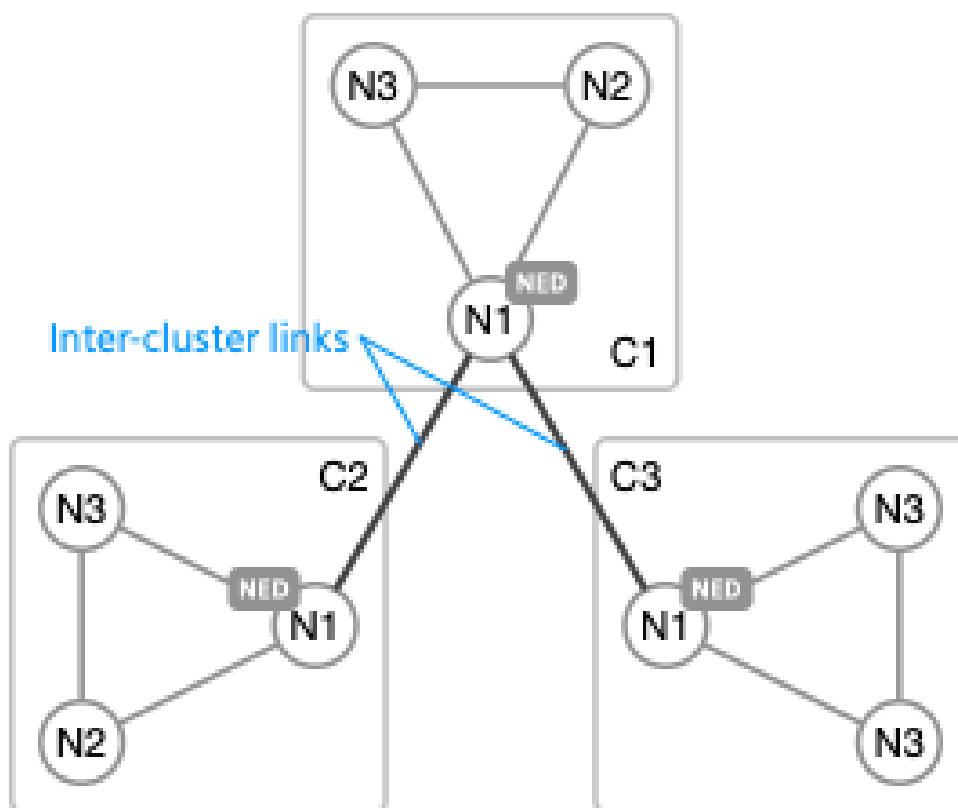


Figure 32: Topology of the infrastructure for the example placement.

Grant Agreement nr: 101092696

The placement starts with step 1 in the SWM-FC instance in the hub cluster. It orders the clusters in the neighbourhood. In this case the order is C1, C3 and C2. C1 is the first cluster for the placement as there are no node recommendations and C1 is the neighbourhood's first cluster. C3 is the second cluster as its link to C1 is faster than the link between C1 and C2. It is *not* checked in this step that the link between C1 and C2 is not fast enough to support channel c12.

The placement continues with step 2 that iterates through the neighbourhood's cluster. The SWM-FC instance in C1 solves the L2 problem shown below. It places workload app•w1 on node C1•N2 and app•w2 on C3. This yields a latency of 1 ms for channel c12 within C1 (i.e., from C1•N2 to C1•N1 which is the NED). In contrast to a problem for a single cluster, the L2 problem contains nodes for clusters C2 and C3, the links of the inter-cluster network, and paths between the nodes in C1 and the cluster nodes.

```
workload assignment v2
model example-L2-problem-in-C1
application app
workload w1 needs cpu:500 ram:1GiB
workload w2 needs cpu:500 ram:1GiB
channel c12:w1->w2 needs lat:10ms bw:10Mb

infrastructure
node N1 provides ram:2GiB cpu:400 // NED
node N2 provides ram:3GiB cpu:900
node N3 provides ram:2GiB cpu:400
cluster C2,C3
  provides cpu:1700 max-cpu:900 ram:7GiB max-ram:3GiB
  with lat:1ms gw-lat:0ms bw:600Mb max-bw:100MiB

network l2sm
link lN1N2,lN2N1:N1<->N2, lN1N3,lN3N1:N1<->N3, lN2N3,l32:N2<->N3
  with lat:1ms bw:100Mb
link lC2C2:C2->C2, lC3C3:C3->C3
  with lat:1ms bw:600Mb
link lN1C2,lC2N1:N1<->C2 with lat:10ms bw:10Mb
link lN1C3,lC3N1:N1<->C3 with lat:5ms bw:10Mb
paths (
  pN1N2:lN1N2 pN1N3:lN1N3 pN2N3:lN2N3 // paths in C1
  pN2N1:lN2N1 pN3N1:lN3N1 pM3N2:lN3N2
  pC2C2:LC2C2 // path in C2
  pN1C2:lN1C2 pN2C2:lN2N1,lN1C2 pN3C2:lN3N1,lN1C2 // paths to C2
  pC2N1:lC2N1 pC2N2:lC2N1,lN1N2 pC2N3:lC2N1,lN1N3
  pC3C3:LC3C3 // path in C3
  pN1C3:lN1C3 pN2C3:lN2N1,lN1C3 pN3C3:lN3N1,lN1C3 // paths to C3
  pC3N1:lC3N1 pC3N2:lC3N1,lN1N2 pC3N3:lC3N1,lN1N3)
```

```
assignment
workload app•w1 on N2
workload app•w2 on C3
channel c12 on l2sm•pN2C3 with lat:1ms
```

With the solution from cluster C1, SWM-FC proceeds to solve the L2 problem in C3. Workload app•w1 is fixed within cluster C1 and cannot be assigned to a node in C3. The inter-cluster network does not have a link between C2 and C3, so C2 is not part of this assignment problem. SWM-FC assigns workload app•w2 to node N2. Channel c12 between workloads app•w1 and app•w2 is now complete (both workloads placed) and fulfils the specified requirements.

```
workload assignment v2
model example-l2-problem-in-C3
application app
workload w1 on C1
workload w2 needs cpu:500 ram:1GiB
channel c12:w1->w2 with lat:1ms needs lat:10ms bw:10Mb

infrastructure
node N1 provides ram:2GiB cpu:400 // NED
node N2 provides ram:3GiB cpu:900
node N3 provides ram:2GiB cpu:400
cluster C1
  provides cpu:1200 max-cpu:400 ram:6GiB max-ram:2GiB
  with lat:1ms gw-lat:0ms bw:600Mb max-bw:100MiB

network l2sm
link lN1N2,lN2N1:N1<->N2, lN1N3,lN3N1:N1<->N3, lN2N3,lM3N2:N2<->N3
  with lat:1ms bw:100Mb
link lC1C1:C1->C1
  with lat:1ms bw:600Mb
link lN1C1,lC1N1:N1<->C1 with lat:5ms bw:10Mb
paths (
  pN1N2:lN1N2 pN1N3:lN1N3 pN2N3:lN2N3 // paths in C3
  pN2N1:lN2N1 pN3N1:lN3N1 pM3N2:lN3N2
  pC1C1:LC1C1 // path in C2
  pN1C1:lN1C1 pN2C1:lN2N1,lN1C1 pN3C1:lN3N1,lN1C1 // paths to C1
  pC1N1:lC1N1 pC1N2:lC1N1,lN1N2 pC1N3:lC1N1,lN1N3)

assignment
workload app•w2 on N2
channel c12 on l2sm•cp2C1
```

SWM-FC placed all application workloads in the neighbourhood. In this case it is not necessary to visit cluster C2 or perform the second round. The instance in the hub cluster tells the instances in the managed clusters to implement the solution found.

3.5.1.3.2 Alternatives

SWM-FC's method of placing workload in a neighbourhood is just one solution to the placement problem. It is a rather simple approach that does not probe clusters and keeps cluster internals private. When splitting up the federated placement problem into pieces like shown, it is not possible to find an optional solution that considers all clusters. This section lists some other methods that may be used. This list is not comprehensive.

- **Avoid multiple problems:** If the neighbourhood and the clusters are small, it might be possible to place workloads by looking at all resources at the same time. This solution does not keep the internal structure private and is feasible only for small environments. On the bright side, this approach allows an optimizer to find an optimal solution given an appropriate target function.
- **Reserve resource immediately:** SWM-FC allocates resources only after a proper placement is found. Reserving the resources may take some time. If placements succeed in the majority of cases or overall time for a placement should be reduced, SWM-FC could allocate resources while working on a placement. If a placement fails, any resources allocated must be disposed of. This solution reduces the total time needed for a placement but might leak resources in case of some errors.
- **Solve L2 problems in parallel:** With this approach the initial L2 problem is given to all clusters that determine a solution. The SWM-FC instance in the hub clusters combines the solutions to come up with a solution for the whole problem. Combining the partial solutions is complex and might take some time. This solution might be faster due to the parallel work in the managed clusters but could trigger too much work in managed clusters. It is hard to guess how long it takes to build a complete solution from the partial solutions.
- **Distributed placement:** SWM-FC has a central component in the hub cluster orchestrating a placement in a neighbourhood. If the SWM-FC instance given a multi-cluster specification performs the placement's orchestration, there is no need to forward the request to the hub cluster. This solution is completely distributed but is more complex as some administrative details are harder to implement when a component like OCM is not available. It also requires that some information that is now only kept on the hub is replicated to all clusters.

3.5.1.4 Interaction with Other Components

SWM-FC provides the same interface as SWM, i.e., clients specify the desired deployment in custom resources. SWM-FC does not collect or determine information about clusters or networks, it only makes use of it. It is agnostic to where the information comes from. SWM-FC does not directly trigger other CODECO components except for NetMA to establish network links that NetMA is already aware of.

SWM-FC reacts to changes to an application's specification or the clusters' infrastructure. PDLC can trigger workload migration by changing an application's node recommendations. This happens in a managed cluster and SWM-FC forwards the change to the hub's SWM-FC instance.

NetMA can update the topology of a network (e.g., the available bandwidth of a link) and this might cause a workload migration as an application's requirements are no longer met. SWM-FC forwards a change by NetMA to its hub instance in the same way as for a change by PDLC.



When placing workloads in a multi-cluster environment, the different SWM-instances have to communicate. Communication is needed while placing workloads, to forward multi-cluster specifications and changes to them to the SWM-FC instance running in the hub cluster, and to keep the hub cluster informed that a managed cluster is reachable. SWM-FC uses one endpoint in each cluster for its communication. SWM-FC does not configure that endpoint itself.

SWM-FC by default does not encrypt traffic between clusters but supports appropriate configuration to enable TLS as supported by Go's standard library. Configuring encryption is easy and allows operators to tailor SWM-FC to their particular environment. For single-cluster deployments, TLS is not necessary.

3.5.2 Sub-components

SWM-FC has the same components as SWM. The orchestrator for multi-cluster placements is not a separate component but built into SWM-FC's main controller. This applies also to the controller tracking reachability of the managed clusters.

- **Extended K8s API and CODECO CRs.** The interface in SWM-FC is the same as that of SWM, i.e., it uses CRs to communicate with other components. There is one new resource types with information about a cluster. Resources defined by SWM are extended to support multi-cluster placements.
- **QoS model.** SWM-FC has the same model for applications, workload, channels and the infrastructure as SWM.
- **SWM controller-manager.** The controllers of SWM-FC executing a placement in a cluster are the same as for SWM. The controller for groups is extended for federated scheduling. There is one additional controller for clusters.
- **QoS Scheduler.** SWM-FC reuses SWM's QoS scheduler unchanged for doing deployments in a managed cluster.
- **Workload Placement Solver.** SWM-FC extends SWM's interface for workload placement solvers to handle multi-cluster placements. The interface has some new fields (e.g., the neighbourhood in a group) but is essentially the same. There is one new type of node to represent clusters when solving L2 problems. SWM-FC's solver is the one from SWM adapted to the new interface. It is deployed as a pod. The more sophisticated optimizer is adapted for SWM-FC and is available for CODECO partners as a container image.
- **Federated Placer.** This component (technically part of SWM's controller manager) implements SWM-FC's method for placing workloads in a neighbourhood. The placer's interface between SWM-FC's instances in the hub and managed clusters is an extension of the interface to the placement solver. The extension is concerned with forwarding specification and changes to the hub cluster and implementing and rolling back a federated assignment in a managed cluster.

SWM-FC supports the extension mechanism of SWM. The solver or optimizer provided by SWM-FC may be replaced by a custom component that implements the interface for a solver. SWM-FC uses this interface when solving L2 problems, i.e., a custom solver can take part in federated scheduling.

SWM-FC's placer reuses SWM components. When implementing a placement in a cluster, it creates a single-cluster assignment. The controller for an assignment plan is adapted to handle some special cases (e.g., channel between clusters).

3.5.2.1 Changes to the Custom Resources

SWM-FC reuses the CRDs of SWM. It adds some fields needed for multi-cluster placements. A CR for SWM-FC is still a valid CR for SWM with some unknown fields as far as SWM is concerned. If a resource of SWM-FC is given to SWM, SWM treats it as a single-cluster CR.

- **Application Group.** A group's placement starts when there are enough applications for the group and there is information for all clusters in the neighbourhood available.
 - *neighbourhood:* A list of cluster names. After placing has begun, the neighbourhood must not be changed.
 - *single-cluster:* A flag indicating whether all workloads of the group must run in the same cluster.
- **Application.**
 - *cluster:* Name of a cluster. If specified, SWM-FC places all workloads of an application in the given cluster. If there is no cluster with this name, the application cannot be placed.
 - *single-cluster:* A flag indicating whether all workloads of the application must run in the same cluster.
- **Cluster.** A cluster resource holds information about a cluster. Cluster resources are only used by SWM-FC's instance running on the hub cluster. A cluster resource contains aggregated and administrative information about a cluster. Resources for clusters are stored in SWM-FC's namespace.

When solving an L2 problem, any information needed for solving it is given in the request. An SWM-FC instance running in a managed cluster needs no cluster resources.

- **NetworkTopology.** SWM-FC uses the same network topology resource used to describe a cluster's network also for the networks connecting clusters. For an inter-cluster network, the "worker" nodes are clusters. SWM-FC uses network names to stitch cluster networks and networks connecting clusters together.
- **Workload.**
 - *cluster:* Name of a cluster. If specified, SWM-FC places the workload in the given cluster. If there is no cluster with this name, the application cannot be placed.

3.5.2.2 Changes To Solver Interface

The interface used to talk to a workload placement solver is extended to include the new fields in the custom resources. In addition, there is a latency field in a channel assignment to return a channel's latency in a cluster.

4 Continuous Integration, Testing, Deployment Preparation and Releasing

This section discusses the final integration approach and the automated processes that help with the development, continuous integration, system testing, deployment preparation, and release of the CODECO Federated Toolkit v2.0. It builds on the best practices for OSS integration and the foundations for CI/CD that were set up during the development of the CODECO Basic Toolkit (D11 and D12). It also finalizes their extension to meet the needs and operational complexity of a federated, multi-cluster orchestration environment.



The official CODECO repository is provided by the Eclipse Foundation via the CODECO Eclipse GitLab. For the testing and integration of the overall framework, the consortium has made use of the infrastructure offered by ICOM, which contributes to the project in two complementary ways, namely through:

- a **dedicated CI/CD server**, linked to the CODECO Eclipse GitLab through **GitLab Runners**, enabling automated unit and system testing;
- a **private container registry**, supporting CI/CD pipelines that generate and manage a large number of intermediate container images, a functionality not provided directly by the Eclipse GitLab infrastructure.

In the final phase of the project, CODECO partners push their code contributions directly to the official **CODECO Eclipse GitLab repository**, where **GitLab Runners are fully integrated** to support **automated testing, building, and deployment** of CODECO components to the connected integration clusters. In addition, the private registry hosted by ICOM continues to be utilised as an intermediate container registry, supporting the execution of the Continuous Integration and Continuous Deployment (CI/CD) pipelines required for the final release of the toolkit.

As described in this section, the established workflow followed during the final integration and release phase is summarised as follows:

- CODECO partners upload their OSS contributions at sub-component level to the official CODECO repository. New commits and merge requests trigger CI/CD pipelines, as defined in the repository's `.gitlab-ci.yml` configuration.
- The ICOM GitLab environment supports continuous integration, enabling consistent and automated integration of CODECO components and interfaces.
- The overall CI/CD process and deployment workflow are detailed in the following subsections, with additional technical details provided in **Section 4.2**.

4.1 Testing Methodology

CODECO followed a continuous integration, system testing, deployment, release methodology, which has been applied throughout the project and finalised for the delivery of the *CODECO Federated Toolkit v2.0*. This methodology ensures that all software components included in the final release have been systematically validated and integrated before publication. The adopted testing methodology consists of the following stages:

- **Feature – Unit Testing:** Unit testing focuses on validating individual internal CODECO sub-components by simulating or mocking the remaining components of the framework. This practice supports the early detection of failures caused by changes in specific code segments or functions and contributes to the overall software robustness. Developers execute unit tests locally within their development environments, while the same tests are automatically invoked by the Continuous Integration (CI) server. Development teams define the necessary CI/CD job configurations using `.gitlab-ci.yml` files to ensure consistent execution during integration.
- **Integration – Testing:** Integration testing verifies the correct operation of services and their features by assessing the interaction between multiple CODECO components and sub-components, while selectively simulating external dependencies. All CODECO components included in the final release are required to successfully pass the defined integration tests prior to publication. Development teams provide instructions on how component interfaces and functionalities can be accessed and exercised during testing, as well as guidance on component deployment. Integration tests are

automatically triggered by the Continuous Integration server when updates to component dependencies are detected.

- **System Testing:** System testing involves the deployment of the complete CODECO framework and the evaluation of its core functionalities and user interactions in realistic environments. The objective of system testing is to assess the compliance of the integrated system with the specified requirements. CODECO components and applications are deployed on decentralised Kubernetes clusters, where experimenters can explicitly define the characteristics of the Kubernetes environments used for testing. System-level tests validate whether predefined functional and operational requirements are met, confirming the readiness of the CODECO Federated Toolkit for final release.

4.2 Deployment and CI/CD Methodology

4.2.1 Deployment

The CODECO OSS Toolkits, including both the Basic (single-cluster) and the Federated (multi-cluster) toolkits, follow a modular design, composed of interoperable yet independently deployable components. Each CODECO component consists of one or more sub-components, which are designed to operate either independently or as part of the integrated CODECO framework.

Each CODECO sub-component is encapsulated in a container image and deployed independently in the form of a Kubernetes deployment object. In cases where a sub-component acts as a Kubernetes controller, its deployment additionally requires the installation of the corresponding Custom Resource Definitions (CRDs). This approach ensures consistency across deployments while enabling fine-grained lifecycle management of individual CODECO components.

Containerisation enables CODECO components and services to be packaged and executed in an isolated and secure manner. This design choice ensures portability across heterogeneous computing environments, independent of the underlying hardware or operating system. For each CODECO sub-component, developers provide a Dockerfile that defines how the corresponding container image is built. These images are used both by contributors during development and testing, as well as by the automated CI/CD pipelines that build, test, and deploy the CODECO services. The CODECO container images included in the final release are published in the public *hecodeco* repository on DockerHub.

To deploy the CODECO OSS Toolkits, a set of Kubernetes distributions has been utilised to support development, integration, testing, and validation of the framework across heterogeneous environments. In particular, the following Kubernetes flavours have been used:

- **KinD (Kubernetes in Docker)**, employed for local development, initial integration activities, and the execution of CI/CD pipelines.
- **Kubeadm-based Kubernetes deployments**, used to validate CODECO operation in production-like environments with physical or virtual nodes, as opposed to container-based nodes.
- **K3s**, adopted as a lightweight Kubernetes distribution supporting arm64 architectures, enabling the deployment of CODECO on Raspberry Pi nodes and other constrained devices, and supporting realistic Edge-Cloud Continuum scenarios.

From a deployment perspective, the key distinction between the Federated Toolkit and the Basic Toolkit is the support for multi-cluster operation. In the final release, CODECO components are fully integrated and deployed in a federated configuration, where multiple Kubernetes clusters are interconnected and coordinated through the Open Cluster Management



(OCM) framework. OCM is used to manage and orchestrate inter-cluster activities, enabling consistent deployment and operation across all participating clusters.

For the federated, multi-cluster environments used during integration and validation, the consortium has made use of KinD-based clusters as well as virtual machines with K3s and OCM hosted within the shared project infrastructure, ensuring reproducibility and stability of the final CODECO toolkit deployment.

4.2.2 CI/CD Methodology

GitLab Runners have been set up by partner ICOM within the CODECO Eclipse GitLab Research project to facilitate the continuous integration and deployment of CODECO components. GitLab Runners are deployed on servers hosted by ICOM as well as on IBM's shared infrastructure, and they interact directly with the CI cluster. The CODECO Eclipse GitLab also utilizes the private container registry deployed on IBM's shared infrastructure, which is essential for constructing container images during CI/CD workflows. The ICOM private GitLab environment is completely aligned with the official CODECO Eclipse GitLab structure, and the source code is available to consortium partners and EC representatives. Within each CODECO sub-component, a Dockerfile is supplied for the purpose of constructing the container images of the CODECO services. These Dockerfiles are utilized by GitLab CI pipelines to autonomously construct images during the integration and testing phases. Developers define the necessary CI/CD tasks, including building, testing, packaging, and deploying artifacts to the integration Kubernetes clusters, through the respective `.gitlab-ci.yml` files. In the case of build or test failures, the CI server alerts the accountable developer, facilitating prompt resolution. The GitLab container registry is also employed to distribute container images across pipeline stages and to facilitate automated deployments.

An example for a `gitlab-ci.yml` file can be found: [here](#)

When the contributor uploads the code to the right CODECO Eclipse GitLab repositories and all of the automated tests pass, they send a merge request to the appropriate sub-component leader (task leader) for evaluation. The changes are added to the main branch of each CODECO project repository after they are approved. A new tag is made and linked to a new code version at the end of each implementation cycle. This version is then published and made available. Only the code that has been validated and is in the main branch is used to make official releases.

Additionally, at the end of each implementation cycle, the container images for the released sub-components are uploaded to the CODECO DockerHub repository (*hecodeco*). Annex II provides more information about the release versions of each sub-component.

4.2.3 Integration and Testing Facilities

The integration and testing activities of the Federated Toolkit are currently conducted on the IBM shared infrastructure, which serves as the main validation environment for multi-cluster CODECO deployments. This infrastructure consists of several virtual machines (VMs) provisioned and maintained within AWS's cloud environment.

Each VM hosts Kubernetes clusters based on k3s, configured to reflect realistic edge-cloud deployment scenarios. Open Cluster Management (OCM) is installed to enable federation capabilities and to support multi-cluster orchestration across the distributed clusters.

In addition to the k3s-based infrastructure, extensive automated testing is performed using kind (Kubernetes-in-Docker) environments deployed on IBM VMs. Typically, three Kubernetes clusters are instantiated to emulate multi-cluster CODECO scenarios. These lightweight environments are primarily used to validate:

- Multi-cluster service deployment



- OCM-based cluster registration and management
- Cross-cluster communication mechanisms
- Policy enforcement

All integration and testing processes are fully automated. Cluster provisioning, component deployment, federation configuration, and validation procedures are executed through automation scripts.

5 Conclusions, Takeaways, Extensions and Sustainability

This deliverable presents the final release (v2.0) of the CODECO OSS Federated Operation Toolkit, consolidating the work carried out in Work Package 4 and completing the transition from single-cluster to federated CODECO. D14 finalises the design, implementation, and integration of the federated components — ACM-FC, PDLC-FC, NetMA-FC, MDM-FC, and SWM-FC — enabling application-centric, context-aware orchestration across heterogeneous Cloud–Edge–IoT environments through Kubernetes-native mechanisms and Open Cluster Management as the federation substrate. System-level integration and validation activities have been conducted on IBM's shared infrastructure, supporting both k3s clusters with OCM and fully automated multi-cluster experimentation using KinD.

5.1 Key Takeaways

D14 delivers a comprehensive and stable set of open-source federated orchestration tools that realise CODECO's design vision for the Edge–Cloud continuum in practice. The following points summarise the most important outcomes of this work:

- **Kubernetes-native federation:** CODECO extends Kubernetes without replacing its mental model. All enforcement remains reconciliation-driven and manifest-based. Federation, neighbourhood scoping, and context-awareness operate beneath the application abstraction.
- **Separation of concerns:** The architectural separation between advisory intelligence (PDLC) and deterministic enforcement (SWM) is a deliberate design choice that ensures policy compliance, auditability, and predictable system behaviour under automated orchestration.
- **Bounded optimisation:** Neighbourhood scoping is fundamental to the scalability of the federated architecture. By restricting monitoring exchange, learning, and scheduling to application-relevant cluster subsets, CODECO avoids global state explosion while preserving adaptability.
- **Multi-dimensional observability:** Energy, network, compute, and data metrics are treated as first-class inputs to the orchestration pipeline, enabling placement decisions that reflect operational reality rather than relying on simplified resource proxies.
- **Open-source and extensible:** The toolkit is fully open-source, modular, and designed for extension. Custom solvers, connectors, cost profiles, and monitoring exporters can be integrated without modifying core components.
- **TRL 4–5 maturity:** The final release has reached the technology readiness level set out in the project work plan, validated through automated multi-cluster testing on the CODECO shared cloud infrastructure and forming a stable basis for broader experimentation and community adoption under Eclipse KuDECO.

5.2 Sustainability

CODECO's federated architecture has been designed with sustainability as a first-class concern, rather than a post-hoc addition. Several design decisions directly support energy-aware and environmentally conscious orchestration across the CEI continuum.

Energy monitoring is implemented as a multi-level model spanning node, link, and flow granularities. Node energy reflects compute activity per cluster, link energy captures the cost of inter-node traffic, and flow energy isolates the energy footprint of individual application data flows. These three dimensions are collected passively via eBPF-based monitoring, meaning all energy estimates are grounded in real observed traffic rather than synthetic probes. This ensures that energy-aware scheduling decisions are based on operational conditions rather than modelled approximations.

At the scheduling layer, energy is treated as a first-class optimisation parameter. The PDLC-CA component supports a pre-defined Greenness performance profile, which weights energy metrics in the composite cost scoring used to rank candidate nodes and clusters. Developers can also define custom profiles that incorporate CO₂ foot printing or other sustainability indicators using normalised CODECO metrics and user-supplied Prometheus queries.

The separation between advisory intelligence and deterministic enforcement means that sustainability objectives can be made measurable, comparable, and enforceable. Placement decisions driven by greenness profiles are policy-filtered and reconciliation-based, ensuring that energy optimisation does not bypass governance or compliance constraints.

Looking ahead, the transition to Eclipse KuDECO provides a community governance structure under which sustainability-oriented extensions such as carbon-aware scheduling, renewable energy signal integration, or broader energy accounting models can be developed and maintained as part of the open-source project. The current implementation establishes the instrumentation and scoring foundation necessary for these extensions to be built upon without requiring architectural changes to the core framework.

5.3 Open Issues

A number of known limitations remain in the current release that developers and integrators should be aware of:

- **PDLC Synchronizator:** The database replication module within PDLC-DP has been fully implemented and functionally verified, but is not yet deployed in production mode. It is currently considered at an advanced prototype level.
- **PDLC-DL-GNN integration:** The GNN sub-component can be used standalone but is not yet fully integrated into the PDLC pipeline as a first-class component. Its outputs are consumed by PDLC-MARL when available, but the integration path is not yet automated end-to-end.
- **PDLC-MARL multi-cluster training:** Multi-cluster training for MARL agents is not yet implemented. The current multi-cluster capability is limited to inference and the auction-based Q-value sharing protocol. Training remains a single-cluster operation.
- **SWM-FC inter-cluster network paths:** The inter-cluster network layer does not support paths spanning more than one link. Clusters that are not directly connected cannot be used to route inter-workload traffic, which limits topology flexibility in larger federations.
- **Secure Connectivity (L2S-M):** The Secure Connectivity sub-component of NetMA-FC is operational but its full design is still being disclosed. Some advanced overlay configuration capabilities remain under development.

5.4 Future Improvements and Extensions

The current implementation provides a solid foundation for federated orchestration, with several directions identified for future development:

- **PDLC-MARL distributed training:** Extending the MARL framework to support multi-cluster online training is a natural next step, enabling agents to jointly learn placement policies across the federation rather than only sharing inference-time Q-values.
- **GNN model adaptability:** The pre-trained STGNN models currently cover cluster topologies of two to eight nodes. Expanding the model registry and automating re-training triggers for unseen topologies would improve operational coverage.
- **Neighbourhood recompositing:** The current neighbourhood scoping is fixed at deployment time and only periodically re-evaluated. Dynamic recompositing in response to cluster failures, load shifts, or mobility events would improve adaptability.
- **SWM-FC multi-hop inter-cluster routing:** Extending the inter-cluster network model to support multi-hop paths would enable more flexible placement across larger and more complex federated topologies.
- **Extended MDM metadata model:** Additional connectors for domain-specific data systems — such as data lakes, streaming platforms, or compliance registries — would broaden the metadata coverage available to the scheduling layer.
- **Sustainability extensions:** The transition to Eclipse KuDECO provides a community governance structure under which carbon-aware scheduling, renewable energy signal integration, and broader energy accounting models can be developed on top of the instrumentation and scoring foundation established in this release.
- **Eclipse KuDECO:** More broadly, the ongoing transition to Eclipse KuDECO will bring community governance, long-term maintenance, and extended adoption beyond the project's lifecycle.

6 References

- [1] Karageorgos, K., et al, (2025). CODECO D13: CODECO Federated Operation and Toolkit v1.0. Zenodo. <https://doi.org/10.5281/zenodo.17412330>
- [2] R. C. Sofia et al. (2026). D31: CODECO Federated Cluster Operation architectural Design. March 2026.
- [3] R. C. Sofia et al (2026). Towards Scalable Federated Container Orchestration: the CODECO approach. ArXiv pre-print (under submission). Jan 2026. <https://arxiv.org/abs/2601.13351>
- [4] R. C. Sofia et al. (2024). CODECO Deliverable D10 - Technological Guidelines, Reference Architecture, and Open-source Ecosystem Design. Zenodo. <https://doi.org/10.5281/zenodo.12819444>
- [5] L. M. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *Proceedings of IEEE INFOCOM 2001 – Conference on Computer Communications*, vol. 3, 2001, pp. 1548–1557.



7 Annex I: CODECO Metrics

This annex provides a revision of the CODECO cross-layer metrics presented in Deliverable D13. A description of the metrics presented is as follows:

- Table 21 presents the minimum subset of parameters considered in CODECO that define an application – [CAM parameters](#). Examples for CAM formulations are provided in the CODECO Eclipse GitLab. The [full schema](#) for the CAM is also provided in the GitLab, in [acm/config/crd/bases/codeco.he-codeco.eu_codecoapps.yaml](#).
- Table 22 describes the metrics collected in MDM, and available via Prometheus. The [schema for MDM](#) is available in the gitlab.
- Table 23 describes the metrics collected in NetMA, and available via Prometheus.

Table 21: CODECO Application Model Attributes, spec, and status, minimum subset of parameters.

Attribute	Former (D10)	Description	Values	Units
<i>Spec</i>				
<i>Uid</i>	<i>Userld</i>	User ID (UID) typically refers to the Linux user ID under which a containerized process runs inside a Pod. This is a security-related setting, not a Kubernetes-specific user concept (like in RBAC), but rather the UID inside the container runtime environment. <i>Available in K8s (runasUser), but should be provided by other CODECO components by ACM</i>	<i>Positive integer; 0 not to be used (root); 1000, first regular user in UNIX; 1001 and above for non-root users.</i>	<i>n.a.</i>
<i>Gid</i>	<i>Usergroup</i>	<i>When dealing with Kubernetes Role-Based Access Control (RBAC), user groups are used to define authorization rules for groups of users</i>	<i>Available in K8s (runasGroup), should be made available if required.</i>	

Attribute	Former (D10)	Description	Values	Units
		<i>who interact with the Kubernetes API server. Identification of the group of user DEV</i>		
<i>performanceProfile</i>	<i>targetProfile</i>	<i>Desired performance profile for the application. CODECO defines greenness, resilience, UserDefined</i>	<i>Greenness, Resilience, UserDefined (string)</i>	<i>n.a.</i>
<i>qosClass</i>	<i>qos</i>	<i>Preferred QoS level. A scale 1-5 is provided to the user and converted to a desired QoS model, e.g., Diffserv co-depends.</i>	<i>Gold, Silver, BestEffort²¹ string</i>	<i>n.a.</i>
<i>securityClass</i>	<i>securityClass</i>	<i>Desired level of security based on a scale of 1-5</i>	<i>High, Good, Medium, Low, None (string)</i>	<i>n.a.</i>
<i>appEnergyLimit</i>	<i>appEnergyLimit</i>	Maximum desired level of energy expenditure for the overall k8s infrastructure associated with an application (percentage).		W (J/s)
<i>complianceLevel</i>	<i>complianceclass</i>	expected level of compliance, based on a scale of 1-5. Checked by MDM.	<i>Positive integer, 1.5</i>	<i>n.a.</i>
Status				
<i>avgAppCpu</i>	<i>avgAppCPU</i>	Aggregation of the CPU usage of all the services in the app (in vCPU units) Provided by ACM	<i>String-encoded value in cores, e.g., 0.75</i>	<i>Millicores, vCPU = millicores / 1000</i>
<i>avgAppMemory</i>	<i>avgAppMemory</i>	Aggregation of the memory usage of	<i>String</i>	<i>MiB</i>

²¹ The next version will map with the usual Diffserv classes: AF, EF, CS, and EXP.

Attribute	Former (D10)	Description	Values	Units
		all the services in the app		
<i>numPods</i>	<i>numPods</i>	The number of the pods instantiated for the application	<i>Positive integer</i>	<i>n.a.</i>
<i>avgServiceCpu</i>	<i>avgServiceCpu</i>	Average CPU usage per micro service pod	<i>String-encoded value in cores, e.g., 0.75</i>	<i>Millicores, vCPU = millicores / 1000</i>
<i>avgServiceMemory</i>	<i>avgServiceMemory</i>	Average memory usage per micro service pod	<i>String</i>	<i>MiB</i>
<i>clusterName</i>	<i>n.a.</i>	Cluster name (unique, assigned by K8s)	<i>String</i>	<i>n.a.</i>
<i>nodeName</i>	<i>nodeName</i>	Node name (unique, assigned by K8s)	<i>String</i>	<i>n.a.</i>
<i>podName</i>	<i>podName</i>	Pod name (unique, assigned by K8s)	<i>String</i>	<i>n.a.</i>
<i>serviceName</i>	<i>serviceName</i>	Service name, user assigned in the app model	<i>String</i>	<i>n.a.</i>
<i>errorMsg</i>	<i>n.a.</i>	CODECO application error message	<i>String</i>	<i>n.a.</i>
<i>avgNodeCpu</i>	<i>avgNodeCpu</i>	Avg CPU consumption of the application on this node	<i>string-encoded value in cores, e.g., 0.75</i>	<i>Millicores, vCPU = millicores / 1000</i>
<i>avgNodeEnergy</i>	<i>avgNodeEnergy</i>	Avg energy consumption of the node	<i>string</i>	<i>mJ</i>
<i>avgNodeFailure</i>	<i>avgNodeFailure</i>	Node failures averaged over time (EMA)	<i>String (positive integer)</i>	<i>n.a.</i>
<i>avgNodeMemory</i>	<i>avgNodeMemory</i>	Avg memory consumption of the application on this node	<i>string</i>	<i>MiB</i>

Table 22: CODECO metrics being collected by MDM

ID (attribute)	Former (D10)	Description	Values	Units
<i>nodeName</i>		identifier of the node in K8s	String	All
<i>freshness</i>		Age of data since last change	String (numeric)	<i>n.a.</i>

ID (attribute)	Former (D10)	Description	Values	Units
			between 0 and 1)	
Compliance_state	Compliance	<p>The compliance_state schema describes the compliance status of a resource (like a Pod or Deployment) within a system. It includes the source system and regulation under evaluation, the name and type of the resource, and an overall status (e.g., "compliant", "non-compliant"). It also lists detailed controls, each with an ID, status (like "passed" or "failed"), and optional severity and description.</p> <p>All fields are textual—there are no numeric values or units. The schema is designed to represent qualitative compliance information, not metrics.</p>	String	n.a.

Table 23: CODECO metrics being collected by NetMA.

ID (attribute)	Former (D10)	Description	Values	Units
nodeName		Identifier of the node in K8s	String	n.a.
name (inside link)		Identifier of the link of a node	String	n.a.
NSM: netma-nsm-mon (underlay metrics at a node level)				
uLinkFailure		Number of failures of a link	String (integer)	n.a.
uPacketLoss		Amount of packet losses in a connection.	String (float)	
uNodeNetFailure		Sum of elink_failure and link_failure	String (integer)	n.a.
uNodeBandWidth		Egress bandwidth for a node - sum of bw for all Egress links on that node	String (integer)	Bytes, e.g., KB, MB
uNodeDegree		In a network, the degree of a node is defined as the sum of all edges connected to it (later, ingress and egress links)	String (integer)	n.a.

ID (attribute)	Former (D10)	Description	Values	Units
uLatencyNanos		Latency (EMA) for all egress links of a worker node	String (float)	ns
uLinkenergy		The average expenditure of energy over a time window (EMA) for a specific link	String (float)	W (J/s)
NSM (netma-nsm-npp) and Secure Connectivity				
oNodeBand-Width		Ingress bandwidth for a node - sum of bw for all ingress links on that node	String (integer)	B, KB, MB
oNodeDegree		Sum of all Channels for a source node	String (integer)	n.a.
oLatencyNanos/		average latency derived from elink latency; can be based on RTT	String (integer)	ns
uPathLength		Number of hops traversed; Average Shortest Path Length	String (integer)	n.a.

8 Annex II – Release Versioning

Table 24 provides a list of the current CODECO sub-components, URLs for open-source code, and images in Docker. The notation in the Table is as follows:

- **C**: CODECO component acronym.
- **SC**: Sub-component abbreviation.
- **URL**: Source code repository on the CODECO Eclipse GitLab (federated cluster branch).
- **OSS**: Published open-source release (v4.0.0) on the CODECO Eclipse GitLab.

Table 24: CODECO sub-components, URLs, and open-source releases.

C	SC	URL	OSS
ACM	ACM Operator	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/tree/ACM-FC?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/releases/4.0.0
PDLC	PDLC-DL-GNNs	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/PDLC-DL/pdlc-gnns	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/PDLC-DL/pdlc-gnns/-/releases/4.0.0
	PDLC-MARL	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/PDLC-DL/pdlc-rl/-/tree/pdlc-MARL-FC?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/PDLC-DL/pdlc-rl/-/releases/4.0.0
	PDLC-CA	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/pdlc-ca/-/tree/PDLC-CA-FC?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/pdlc-ca/-/releases/4.0.0
	PDLC-DP	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/pdlc-dp/-/tree/PDLC-DP-FC?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/pdlc-dp/-/releases/4.0.0
NetMA	Nemesys FC	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/nemesys-fc	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/nemesys-fc/-/releases/4.0.0
	Network State Management	https://gitlab.eclipse.org/eclipse-research-labs/codeco-	https://gitlab.eclipse.org/eclipse-research-labs/codeco-

C	SC	URL	OSS
		project/network-management-and-adaptation-netma/network-state-management	project/network-management-and-adaptation-netma/network-state-management/-/releases/4.0.0
MDM	Connectors	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/connectors	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/connectors/-/tags/4.0.0
	Graphdb	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb/-/releases/4.0.0
	MDM API	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/mdm-api	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/mdm-api/-/releases/4.0.0
SWM	QoS Scheduler	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/scheduling-and-workload-migration-swm/qos-scheduler	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/scheduling-and-workload-migration-swm/qos-scheduler/-/releases/4.0.0
	Workflow Placement Solver	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/scheduling-and-workload-migration-swm/workload-placement-solver	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/scheduling-and-workload-migration-swm/workload-placement-solver/-/releases/4.0.0

[6]

